Programming and Reasoning with Side-Effects in IDRIS

Edwin Brady

3rd June 2014

Contents

1 Introduction

Pure functional languages with dependent types such as IDRIS (`http://idris-lang.org/`) 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. Realistically, though, things are not so simple: programs have to interact with the outside world, with user input, input from a network, mutable state, and so on. In this tutorial I will introduce the `effects` library, which is included with the IDRIS distribution and supports programming and reasoning with side-effecting programs, supporting mutable state, interaction with the outside world, exceptions, and verified resource management.

This tutorial assumes familiarity with pure programming in IDRIS, as described in Sections 1–6 of the main tutorial [9][1]. The examples are tested with IDRIS version 0.9.12.

Consider, for example, the following introductory function which illustrates the kind of properties which can be expressed in the type system:

```
vadd : Vect n Int -> Vect n Int -> Vect n Int
vadd []        []        = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

This function adds corresponding elements in a pair of vectors. The type guarantees that the vectors will contain only elements of type `Int`, and that the input lengths and the output length all correspond. A natural question to ask here, which is typically neglected by introductory tutorials, is "How do I turn this into a program?" That is, given some lists entered by a user, how do we get into a position to be able to apply the `vadd` function? Before doing so, we will have to:

- Read user input, either from the keyboard, a file, or some other input device.
- Check that the user inputs are valid, i.e. contain only `Int`s and are the same length, and report an error if not.
- Write output

The complete program will include side-effects for I/O and error handling, before we can get to the pure core functionality. In this tutorial, we will see how IDRIS supports side-effects. Furthermore, we will see how we can use the dependent type system to *reason* about stateful and side-effecting programs. We will return to this specific example later.

1.1 Hello world

To give an idea of how programs with effects look in IDRIS, here is the ubiquitous "Hello world" program, written using the `effects` library:

```
module Main

import Effect.StdIO

hello : { [STDIO] } Eff IO ()
hello = putStrLn "Hello world!"

main : IO ()
main = run hello
```

[1] You do not, however, need to know what a monad is. A correctness property of this tutorial is that the word "monad" should appear exactly twice, both in this footnote.

As usual, the entry point is `main`. All `main` has to do is invoke the `hello` function which supports the `STDIO` effect for console I/O, runs in the `IO` context, and returns the unit value. The details of the `Eff` type will be presented in the remainder of this tutorial.

To compile and run this program, IDRIS needs to be told to include the `effects` package, using the `-p effects` flag (this flag is required for all examples in this tutorial):

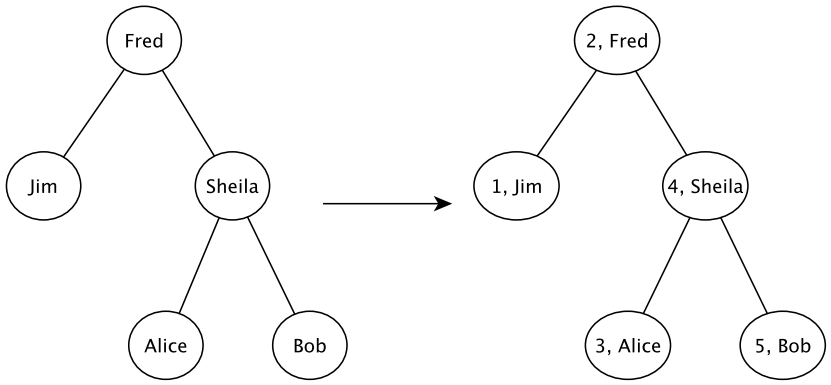
```
$ idris hello.idr -o hello -p effects
$ ./hello
Hello world!
```

1.2 Outline

The tutorial is structured as follows: first, in Section 2, we will discuss state management, describing why it is important and introducing the `effects` library to show how it can be used to manage state. This section also gives an overview of the syntax of effectful programs. Section 3 then introduces a number of other effects a program may have: I/O; Exceptions; Random Numbers; and Non-determinism, giving examples for each, and an extended example combining several effects in one complete program. Section 4 introduces *dependent* effects, showing how states and resources can be managed in types. Section 5 shows how new effects can be implemented. Section 6 gives an extended example showing how to implement a "mystery word" guessing game, using effects to describe the rules of the game and ensure they are implemented accurately. References to further reading are given in Section 7.

2 State

Many programs, even pure programs, can benefit from locally mutable state. For example, consider a program which tags binary tree nodes with a counter, by an inorder traversal (i.e. counting depth first, left to right). This would perform something like the following:



We can describe binary trees with the following data type `BTree` and `testTree` to represent the example input above:

```
data BTree a = Leaf
             | Node (BTree a) a (BTree a)

testTree : BTree String
testTree = Node (Node Leaf "Jim" Leaf)
```

```
                "Fred"
                (Node (Node Leaf "Alice" Leaf)
                      "Sheila"
                      (Node Leaf "Bob" Leaf))
```

Then our function to implement tagging, beginning to tag with a specific value `i`, has the following type:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
```

First attempt

Naïvely, we can implement `treeTag` by implementing a helper function which propagates a counter, returning the result of the count for each subtree:

```
treeTagAux : (i : Int) -> BTree a -> (Int, BTree (Int, a))
treeTagAux i Leaf = (i, Leaf)
treeTagAux i (Node l x r)
       = let (i', l') = treeTagAux i l in
         let x' = (i', x) in
         let (i'', r') = treeTagAux (i' + 1) r in
             (i'', Node l' x' r')

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = snd (treeTagAux i x)
```

This gives the expected result when run at the IDRIS REPL prompt:

```
*TreeTag> treeTag 1 testTree
Node (Node Leaf (1, "Jim") Leaf)
     (2, "Fred")
     (Node (Node Leaf (3, "Alice") Leaf)
           (4, "Sheila")
           (Node Leaf (5, "Bob") Leaf)) : BTree (Int, String)
```

This works as required, but there are several problems when we try to scale this to larger programs. It is error prone, because we need to ensure that state is propagated correctly to the recursive calls (i.e. passing the appropriate `i` or `i'`). It is hard to read, because the functional details are obscured by the state propagation. Perhaps most importantly, there is a common programming pattern here which should be abstracted but instead has been implemented by hand. There is local mutable state (the counter) which we have had to make explicit.

2.1 Introducing `effects`

IDRIS provides a library, `effects` [2], which captures this pattern and many others involving effectful computation[2]. An effectful program `f` has a type of the following form:

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

That is, the return type gives the effects that `f` supports (`effs`, of type `List EFFECT`), the *computation context* it runs in (`m`) and the type the computation returns `t`. The computation context can be any function of type `Type -> Type`, for example `id`. When we come to run an effectful program in `{ effs } Eff m t`, the result will be of type `m t`. So, our `treeTagAux` helper could be written with the following type:

```
treeTagAux : BTree a -> { [STATE Int] } Eff id (BTree (Int, a))
```

[2]The earlier paper [2] describes the essential implementation details, although the library presented there is an earlier version which is less powerful than that presented in this tutorial.

That is, `treeTagAux` has access to an integer state, because the list of available effects includes `STATE Int`. `STATE` is declared as follows in the module `Effect.State` (that is, we must `import Effect.State` to be able to use it):

```
STATE : Type -> EFFECT
```

It is an effect parameterised by a type (by convention, we write effects in all capitals). The `treeTagAux` function runs in the identity computation context and returns a new tree tagged with `Int`s. It is implemented as follows:

```
treeTagAux Leaf = pure Leaf
treeTagAux (Node l x r)
    = do l' <- treeTagAux l
         i <- get
         put (i + 1)
         r' <- treeTagAux r
         pure (Node l' (i, x) r')
```

There are several remarks to be made about this implementation. Essentially, it hides the state, which can be accessed using `get` and updated using `put`, but it introduces several new features. Specifically, it uses `do`-notation, binding variables with `<-`, and a `pure` function. There is much to be said about these features, but for our purposes, it suffices to know the following:

- `do` blocks allow effectful operations to be sequenced.
- `x <- e` binds the result of an effectful operation `e` to a variable `x`. For example, in the above code, `treeTagAux l` is an effectful operation returning a pair `(Int, BTree a)`, so `l'` has type `(Int, BTree a)`.
- `pure e` turns a pure value `e` into the result of an effectful operation.

The `get` and `put` functions read and write a state `t`, assuming that the `STATE t` effect is available. They have the following types, polymorphic in the state `t` they manage and the context `m` in which they run:

```
get :      { [STATE t] } Eff m t
put : t -> { [STATE t] } Eff m ()
```

A program in `Eff` can call any other function in `Eff` provided that the calling function supports at least the effects required by the called function. In this case, it is valid for `treeTagAux` to call both `get` and `put` because all three functions support the `STATE Int` effect.

To run a program in `Eff` (that is, to evaluate it in an appropriate context), we use the `run` or `runPure` function. Using `runPure`, which runs an effectful program in the identity context, we can write the `treeTag` function as follows, using `put` to initialise the state:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPure (do put i
                          treeTagAux x)
```

We could also run the program in an impure context, such as `IO`, by changing the type of `treeTagAux` to be polymorphic in computation contexts, and using `run` instead of `runPure`:

```
treeTagAux : BTree a -> { [STATE Int] } Eff m (BTree (Int, a))
...

treeTag : (i : Int) -> BTree a -> IO (BTree (Int, a))
treeTag i x = run (do put i
                      treeTagAux x)
```

Note that the definition of `treeTagAux` is exactly as before. All that has changed is that the type is more generic, and can run in *any* computation context `m`. There are no constraints on `m` other than that it must have type `Type -> Type`. For reference, this complete program (including a `main` to run it) is shown in Listing 1.

Listing 1: Tree tagging

```
module Main

import Effect.State

data BTree a = Leaf
             | Node (BTree a) a (BTree a)

instance Show a => Show (BTree a) where
    show Leaf = "[]"
    show (Node l x r) = "[" ++ show l ++ " "
                            ++ show x ++ " "
                            ++ show r ++ "]"

testTree : BTree String
testTree = Node (Node Leaf "Jim" Leaf)
              "Fred"
              (Node (Node Leaf "Alice" Leaf)
                    "Sheila"
                    (Node Leaf "Bob" Leaf))

treeTagAux : BTree a -> { [STATE Int] } Eff m (BTree (Int, a))
treeTagAux Leaf = pure Leaf
treeTagAux (Node l x r)
  = do l' <- treeTagAux l
       i <- get
       put (i + 1)
       r' <- treeTagAux r
       pure (Node l' (i, x) r')

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPure (do put i
                          treeTagAux x)

main : IO ()
main = print (treeTag 1 testTree)
```

2.2 Effects and Resources

Each effect is associate with a *resource*, which is initialised before an effectful program can be run. For example, in the case of `STATE Int` the corresponding resource is the integer state itself. The types of `runPure` and `run` show this (slightly simplified here for illustrative purposes):

```
runPure : {env : Env id xs} -> { xs } Eff id a -> a
run : Applicative m => {env : Env m xs} -> { xs } Eff m a -> m a
```

The `env` argument is implicit, and initialised automatically where possible using default values given by instances of the following type class:

```
class Default a where
    default : a
```

Instances of `Default` are defined for all primitive types, and many library types such as `List`, `Vect`, `Maybe`, pairs, etc. However, where no default value exists for a resource type (for example, you may want a `STATE` type for which there is no `Default` instance) the resource environment can be given explicitly using one of the following functions:

```
runPureInit : Env id xs -> { xs } Eff id a -> a
runInit : Applicative m => Env m xs -> { xs } Eff m a -> m a
```

To be well-typed, the environment must contain resources corresponding exactly to the effects in `xs`. For example, we could also have implemented `treeTag` by initialising the state as follows:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPureInit [i] (treeTagAux x)
```

2.3 Labelled Effects

What if we have more than one state, especially more than one state of the same type? How would `get` and `put` know which state they should be referring to? For example, how could we extend the tree tagging example such that it additionally counts the number of leaves in the tree? One possibility would be to change the state so that it captured both of these values, e.g.:

```
treeTagAux : BTree a -> { [STATE (Int, Int)] } Eff m (BTree (Int, a))
```

Doing this, however, ties the two states together throughout (as well as not indicating which integer is which). It would be nice to be able to call effectful programs which guaranteed only to access one of the states, for example. In a larger application, this becomes particularly important.

The `effects` library therefore allows effects in general to be *labelled* so that they can be referred to explicitly by a particular name. This allows multiple effects of the same type to be included. We can count leaves and update the tag separately, by labelling them as follows:

```
treeTagAux : BTree a -> { ['Tag ::: STATE Int,
                           'Leaves ::: STATE Int] } Eff m (BTree (Int, a))
```

The `:::` operator allows an arbitrary label to be given to an effect. This label can be any type—it is simply used to identify an effect uniquely. Here, we have used a symbol type. In general `'name` introduces a new symbol, the only purpose of which is to disambiguate values[3].

When an effect is labelled, its operations are also labelled using the `:-` operator. In this way, we can say explicitly which state we mean when using `get` and `put`. The tree tagging program which also counts leaves can be written as follows:

```
treeTagAux Leaf = do 'Leaves :- update (+1)
                     pure Leaf
treeTagAux (Node l x r)
  = do l' <- treeTagAux l
       i <- 'Tag :- get
       'Tag :- put (i + 1)
       r' <- treeTagAux r
       pure (Node l' (i, x) r')
```

[3] In practice, `'name` simply introduces a new empty type

The `update` function here is a combination of `get` and `put`, applying a function to the current state.

```
update : (x -> x) -> { [STATE x] } Eff m ()
```

Finally, our top level `treeTag` function now returns a pair of the number of leaves, and the new tree. Resources for labelled effects are intialised using the `:=` operator (reminisicent of assignment in an imperative language):

```
treeTag : (i : Int) -> BTree a -> (Int, BTree (Int, a))
treeTag i x = runPureInit ['Tag := i, 'Leaves := 0]
                      (do x' <- treeTagAux x
                          leaves <- 'Leaves :- get
                          pure (leaves, x'))
```

To summarise, we have:

- `:::` to convert an effect to a labelled effect.
- `:-` to convert an effectful operation to a labelled effectful operation.
- `:=` to initialise a resource for a labelled effect.

Or, more formally with their types (slightly simplified to account only for the situation where available effects are not updated):

```
(:::) : lbl -> EFFECT -> EFFECT
(:-)  : (l : lbl) -> { [x] } Eff m a -> { [l ::: x] } Eff m a
(:=)  : (l : lbl) -> res -> LRes l res
```

Here, `LRes` is simply the resource type associated with a labelled effect. Note that labels are polymorphic in the label type `lbl`. Hence, a label can be anything—a string, an integer, a type, etc.

2.4 !-notation

In many cases, using `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 the state, for example:

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

This seems unnecessarily verbose, and it would be nice to program in a more direct style in these cases. IDRIS provides `!`-notation to help with this. The above program can be written instead as:

```
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 not really a function, merely syntax! In practice, a subexpression `!expr` will lift `expr` as high 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 y' <- print y
                 x' <- x
                 g' <- g y' x'
                 f g'
```

2.5 Syntactic Sugar and `Eff`

By now, you may be wondering about the syntax we are using for `Eff`, because it doesn't look like a normal IDRIS type! (If not, you may safely skip this section and return to it later.) In fact, the type of `Eff` is the following:

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

This is more general than the types we have been writing so far. It is parameterised over a computation context `m`, a result type `x`, as we have already seen, but also a `List EFFECT` and a function type `x -> List EFFECT`.

These additional parameters are the list of *input* effects, and a list of *output* effects, computed from the result of an effectful operation. That is: running an effectful program can change the set of effects available! This is a particularly powerful idea, and we will see its consequences in more detail later. Some examples of operations which can change the set of available effects are:

- Updating a state containing a dependent type (for example adding an element to a vector).
- Opening a file for reading is an effect, but whether the file really *is* open afterwards depends on whether the file was successfully opened.
- Closing a file means that reading from the file should no longer be possible.

While powerful, this can make uses of the `Eff` type hard to read. Therefore, the `effects` library provides syntactic sugar which is translated such that. . .

```
{ xs } Eff m a
```

. . . is expanded to . . .

```
Eff m a xs (\_ => xs)
```

i.e. the set of effects remains the same on output. This suffices for the `STATE` example we have seen so far, and for many useful side-effecting programs. We could also have written `treeTagAux` with the expanded type:

```
treeTagAux : BTree a ->
             Eff m (BTree (Int, a)) [STATE Int] (\x => [STATE Int])
```

Later, we will see programs which update effects:

```
{ xs ==> xs' } Eff m a
```

. . . which is expanded to . . .

```
Eff m a xs (\_ => xs')
```

i.e. the set of effects is updated to `xs'` (think of a transition in a state machine). There is, for example, a version of `put` which updates the type of the state:

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

Also, we have:

```
{ xs ==> {res} xs' } Eff m a
```

. . . which is expanded to . . .

```
Eff m a xs (\res => xs')
```

i.e. the set of effects is updated according to the result of the operation `res`.

3 Simple Effects

So far we have seen how to write programs with locally mutable state using the `STATE` effect. To recap, we have the definitions in Listing 2 in a module `Effect.State`.

Listing 2: State Effect

```
module Effect.State

STATE : Type -> EFFECT

get    :              { [STATE x] } Eff m x
put    : x ->         { [STATE x] } Eff m ()
putM   : y ->         { [STATE x] ==> [STATE y] } Eff m ()
update : (x -> x) -> { [STATE x] } Eff m ()

instance Handler State m
```

The last line, `instance Handler State m`, means that the `STATE` effect is usable in any computation context `m`. (The lower case `State` is a data type describing the operations which make up the `STATE` effect itself—we will go into more detail about this in Section 5.)

In this section, we will introduce some other supported effects, allowing console I/O, exceptions, random number generation and non-deterministic programming. For each effect we introduce, we will begin with a summary of the effect, its supported operations, and the contexts in which it may be used, like that above for `STATE`, and go on to present some simple examples. At the end, we will see some examples of programs which combine multiple effects.

All of the effects in the library, including those described in this section, are summarised in Appendix A.

3.1 Console I/O

Console I/O (Listing 3) is supported with the `STDIO` effect, which allows reading and writing characters and strings to and from standard input and standard output. Notice that there is a constraint here on the computation context `m`, because it only makes sense to support console I/O operations in a context where we can perform (or at the very least simulate) console I/O.

Examples

A program which reads the user's name, then says hello, can be written as follows:

```
hello : { [STDIO] } Eff IO ()
hello = do putStr "Name? "
           x <- getStr
           putStrLn ("Hello " ++ trim x ++ "!")
```

Listing 3: Console I/O Effect

```
module Effect.StdIO

STDIO : EFFECT

putChar  : Handler StdIO m => Char ->   { [STDIO] } Eff m ()
putStr   : Handler StdIO m => String -> { [STDIO] } Eff m ()
putStrLn : Handler StdIO m => String -> { [STDIO] } Eff m ()

getStr   : Handler StdIO m =>           { [STDIO] } Eff m String
getChar  : Handler StdIO m =>           { [STDIO] } Eff m Char

instance Handler StdIO IO
instance Handler StdIO (IOExcept a)
```

We use `trim` here to remove the trailing newline from the input. The resource associated with `STDIO` is simply the empty tuple, which has a default value `()`, so we can run this as follows:

```
main : IO ()
main = run hello
```

In `hello` we could also use `!`-notation instead of `x <- getStr`, since we only use the string that is read once:

```
hello : { [STDIO] } Eff IO ()
hello = do putStr "Name? "
           putStrLn ("Hello " ++ trim !getStr ++ "!")
```

More interestingly, we can combine multiple effects in one program. For example, we can loop, counting the number of people we've said hello to:

```
hello : { [STATE Int, STDIO] } Eff IO ()
hello = do putStr "Name? "
           putStrLn ("Hello " ++ trim !getStr ++ "!")
           update (+1)
           putStrLn ("I've said hello to: " ++ show !get ++ " people")
           hello
```

The list of effects given in `hello` means that the function can call `get` and `put` on an integer state, and any functions which read and write from the console. To run this, `main` does not need to be changed.

Aside: Resource Types

To find out the resource type of an effect, if necessary (for example if we want to initialise a resource explicitiy with `runInit` rather than using a default value with `run`) we can run the `resourceType` function at the IDRIS REPL:

```
*ConsoleIO> resourceType STDIO
() : Type
*ConsoleIO> resourceType (STATE Int)
Int : Type
```

3.2 Exceptions

Listing 4 gives the definition of the `EXCEPTION` effect, declared in module `Effect.Exception`. This allows programs to exit immediately with an error, or errors to be handled more generally.

Listing 4: Exception Effect

```
module Effect.Exception

EXCEPTION : Type -> EFFECT

raise : a -> { [EXCEPTION a ] } Eff m b

instance           Handler (Exception a) Maybe
instance           Handler (Exception a) List
instance           Handler (Exception a) (Either a)
instance           Handler (Exception a) (IOExcept a)
instance Show a => Handler (Exception a) IO
```

Example

Suppose we have a `String` which is expected to represent an integer in the range `0` to `n`. We can write a function `parseNumber` which returns an `Int` if parsing the string returns a number in the appropriate range, or throws an exception otherwise. Exceptions are paramaterised by an error type:

```
data Err = NotANumber | OutOfRange

parseNumber : Int -> String -> { [EXCEPTION Err] } Eff m Int
parseNumber num str
   = if all isDigit (unpack str)
        then let x = cast str in
             if (x >=0 && x <= num)
                then pure x
                else raise OutOfRange
        else raise NotANumber
```

Programs which support the `EXCEPTION` effect can be run in any context which has some way of throwing errors, for example, we can run `parseNumber` in the `Either Err` context. It returns a value of the form `Right x` if successful:

```
*Exception> the (Either Err Int) $ run (parseNumber 42 "20")
Right 20 : Either Err Int
```

Or `Left e` on failure, carrying the appropriate exception:

```
*Exception> the (Either Err Int) $ run (parseNumber 42 "50")
Left OutOfRange : Either Err Int

*Exception> the (Either Err Int) $ run (parseNumber 42 "twenty")
Left NotANumber : Either Err Int
```

In fact, we can do a little bit better with `parseNumber`, and have it return a *proof* that the integer is in the required range along with the integer itself. One way to do this is define a type of bounded integers, `Bounded`:

```
Bounded : Int -> Type
Bounded x = (n : Int ** so (n >= 0 && n <= x))
```

Recall that `so` is parameterised by a `Bool`, and only `so True` is inhabited. We can use `choose` to construct such a value from the result of a dynamic check:

```
data so : Bool -> Type = oh : so True

choose : (b : Bool) -> Either (so b) (so (not b))
```

We then write `parseNumber` using `choose` rather than an `if/then/else` construct, passing the proof it returns on success as the boundedness proof:

```
parseNumber : (x : Int) -> String -> { [EXCEPTION Err] } Eff m (Bounded x)
parseNumber x str
   = if all isDigit (unpack str)
        then let num = cast str in
             case choose (num >=0 && num <= x) of
                  Left p => pure (num ** p)
                  Right p => raise OutOfRange
        else raise NotANumber
```

If we are going to use this in a larger program, it is likely that we will also want to recover from parse failures. The `effects` library provides the `catch` function for this, taking a program with available effects `xs`, and a function for dealing with any exceptions:

```
catch : Catchable m err =>
        { xs } Eff m a -> (err -> { xs } Eff m a) -> { xs } Eff m a
```

Notice the constraint `Catchable m err`. It is not required that the set of available effects includes `EXCEPTION`, merely that the context supports throwing and catching off errors *independently* of whether a program uses `effects`. It is defined as follows:

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

instance Catchable Maybe ()
instance Catchable (Either a) a
instance Catchable (IOExcept err) err
instance Catchable List ()
```

Using `catch`, we can write a program which takes user input, parses it to check it is a number within a defined range, and uses a default value if there is a parse error. Notice that after reading user input, the resulting proof `prf` is available as a guarantee that the number which has been read from user input is in the range `0-100`.

```
readNum : { [STDIO, EXCEPTION Err] } Eff (IOExcept Err) ()
readNum = do
    putStr $ "Enter a number between 0 and 100:"
    (num ** prf) <- catch (parseNumber 100 (trim !getStr))
                          (\e => do putStrLn $ "FAIL: " ++ show e
                                    return (0 ** oh))
    putStrLn $ show num
```

3.3 Random Numbers

Random number generation is also implemented by the `effects` library, in module `Effect.Random`. Listing 5 gives its definition.

Listing 5: Random Number Effect

```
module Effect.Random

RND : EFFECT

srand  : Integer ->            { [RND] } Eff m ()
rndInt : Integer -> Integer -> { [RND] } Eff m Integer
rndFin : (k : Nat) ->          { [RND] } Eff m (Fin (S k))

instance Handler Random m
```

Random number generation is considered side-effecting because its implementation generally relies on some external source of randomness. The default implementation here relies on an integer *seed*, which can be set with `srand`. A specific seed will lead to a predictable, repeatable sequence of random numbers. There are two functions which produce a random number:

- `rndInt`, which returns a random integer between the given lower and upper bounds.
- `rndFin`, which returns a random element of a finite set (essentially a number with an upper bound given in its type).

Example

We can use the `RND` effect to implement a simple guessing game. The `guess` function, given a target number, will repeatedly ask the user for a guess, and state whether the guess is too high, too low, or correct:

```
guess : Int -> { [STDIO] } Eff IO ()
```

For reference, the code for `guess` is given in Listing 6. Note that we use `parseNumber` as defined previously to read user input, but we don't need to list the `EXCEPTION` effect because we use a nested `run` to invoke `parseNumber`, independently of the calling effectful program.

To invoke these, we pick a random number within the range 0–100, having set up the random number generator with a seed, then run `guess`:

```
game : { [RND, STDIO] } Eff IO ()
game = do srand 123456789
          guess (fromInteger !(rndInt 0 100))

main : IO ()
main = run game
```

If no seed is given, it is set to the `default` value. For a less predictable game, some better source of randomness would be required, for example taking an initial seed from the system time. To see how to do this, see the `SYSTEM` effect described in Appendix A.

3.4 Non-determinism

Listing 7 gives the definition of the non-determinism effect, which allows a program to choose a value non-deterministically from a list of possibilities in such a way that the entire computation succeeds.

Listing 6: Guessing Game

```
guess : Int -> { [STDIO] } Eff IO ()
guess target
    = do putStr "Guess: "
         case run (parseNumber 100 (trim !getStr)) of
              Nothing => do putStrLn "Invalid input"
                            guess target
              Just (v ** _) =>
                            case compare v target of
                                LT => do putStrLn "Too low"
                                         guess target
                                EQ => putStrLn "You win!"
                                GT => do putStrLn "Too high"
                                         guess target
```

Listing 7: Non-determinism Effect

```
import Effect.Select

SELECT : EFFECT

select : List a -> { [SELECT] } Eff m a

instance Handler Selection Maybe
instance Handler Selection List
```

Example

The `SELECT` effect can be used to solve constraint problems, such as finding Pythagorean triples. The idea is to use `select` to give a set of candidate values, then throw an exception for any combination of values which does not satisfy the constraint:

```
triple : Int -> { [SELECT, EXCEPTION String] } Eff m (Int, Int, Int)
triple max = do z <- select [1..max]
                y <- select [1..z]
                x <- select [1..y]
                if (x * x + y * y == z * z)
                   then pure (x, y, z)
                   else raise "No triple"
```

This program chooses a value for `z` between `1` and `max`, then values for `y` and `x`. In operation, after a `select`, the program executes the rest of the `do`-block for every possible assignment, effectively searching depth-first. If the list is empty (or an exception is thrown) execution fails.

There are handlers defined for `Maybe` and `List` contexts, i.e. contexts which can capture failure. Depending on the context `m`, `triple` will either return the first triple it finds (if in `Maybe` context) or all triples in the range (if in `List` context). We can try this as follows:

```
main : IO ()
main = do print $ the (Maybe _) $ run (triple 100)
          print $ the (List _) $ run (triple 100)
```

3.5 vadd revisited

We now return to the `vadd` program from the introduction. Recall the definition:

```
vadd : Vect n Int -> Vect n Int -> Vect n Int
vadd []        []        = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

Using `effects`, we can set up a program so that it reads input from a user, checks that the input is valid (i.e both vectors contain integers, and are the same length) and if so, pass it on to `vadd`. First, we write a wrapper for `vadd` which checks the lengths and throw an exception if they are not equal. We can do this for input vectors of length `n` and `m` by matching on the implicit arguments `n` and `m` and using `decEq` to produce a proof of their equality, if they are equal:

```
vadd_check : Vect n Int -> Vect m Int ->
             { [EXCEPTION String] } Eff e (Vect m Int)
vadd_check {n} {m} xs ys with (decEq n m)
  vadd_check {n} {m=n} xs ys | (Yes refl) = pure (vadd xs ys)
  vadd_check {n} {m}   xs ys | (No contra) = raise "Length mismatch"
```

To read a vector from the console, we implement a function of the following type:

```
read_vec : { [STDIO] } Eff IO (p ** Vect p Int)
```

This returns a dependent pair of a length, and a vector of that length, because we cannot know in advance how many integers the user is going to input. One way to implement this function, using `-1` to indicate the end of input, is shown in Listing 8. This uses a variation on `parseNumber` which does not require a number to be within range.

Finally, we write a program which reads two vectors and prints the result of pairwise addition of them, throwing an exception if the inputs are of differing lengths:

```
do_vadd : { [STDIO, EXCEPTION String] } Eff IO ()
do_vadd = do putStrLn "Vector 1"
             (_ ** xs) <- read_vec
             putStrLn "Vector 2"
             (_ ** ys) <- read_vec
             putStrLn (show !(vadd_check xs ys))
```

By having explicit lengths in the type, we can be sure that `vadd` is only being used where the lengths of inputs are guaranteed to be equal. This does not stop us reading vectors from user input, but it does require that the lengths are checked and any discrepancy is dealt with gracefully.

3.6 Example: An Expression Calculator

To show how these effects can fit together, let us consider an evaluator for a simple expression language, with addition and integer values.

```
data Expr = Val Integer
          | Add Expr Expr
```

An evaluator for this language always returns an `Integer`, and there are no situations in which it can fail!

```
eval : Expr -> Integer
eval (Val x) = x
eval (Add l r) = eval l + eval r
```

If we add variables, however, things get more interesting. The evaluator will need to be able to access the values stored in variables, and variables may be undefined.

Listing 8: Reading a vector from the console

```
read_vec : { [STDIO] } Eff IO (p ** Vect p Int)
read_vec = do putStr "Number (-1 when done): "
              case run (parseNumber (trim !getStr)) of
                   Nothing => do putStrLn "Input error"
                                 read_vec
                   Just v => if (v /= -1)
                                 then do (_ ** xs) <- read_vec
                                         pure (_ ** v :: xs)
                                 else pure (_ ** [])
  where
    parseNumber : String -> { [EXCEPTION String] } Eff m Int
    parseNumber str
      = if all (\x => isDigit x || x == '-') (unpack str)
           then pure (cast str)
           else raise "Not a number"
```

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

To start, we will change the type of `eval` so that it is effectful, and supports an exception effect for throwing errors, and a state containing a mapping from variable names (as `String`s) to their values:

```
Env : Type
Env = List (String, Integer)

eval : Expr -> { [EXCEPTION String, STATE Env] } Eff m Integer
eval (Val x) = return x
eval (Add l r) = return $ !(eval l) + !(eval r)
```

Note that we are using `!`-notation to avoid having to bind subexpressions in a `do` block. Next, we add a case for evaluating `Var`:

```
eval (Var x) = case lookup x !get of
                    Nothing => raise $ "No such variable " ++ x
                    Just val => return val
```

This retrieves the state (with `get`, supported by the `STATE Env` effect) and raises an exception if the variable is not in the environment (with `raise`, supported by the `EXCEPTION String` effect).

To run the evaluator on a particular expression in a particular environment of names and their values, we can write a function which sets the state then invokes `eval`:

```
runEval : List (String, Integer) -> Expr -> Maybe Integer
runEval args expr = run (eval' expr)
  where eval' : Expr -> { [EXCEPTION String, STATE Env] } Eff Maybe Integer
        eval' e = do put args
                     eval e
```

We have picked `Maybe` as a computation context here; it needs to be a context which is available for every effect supported by `eval`. In particular, because we have exceptions, it needs to be a context which supports exceptions. Alternatively, `Either String` or `IO` would be fine, for example.

What if we want to extend the evaluator further, with random number generation? To achieve this, we add a new constructor to `Expr`, which gives a random number up to a maximum value:

```
data Expr = Val Integer
          | Var String
          | Add Expr Expr
          | Random Integer
```

Then, we need to deal with the new case, making sure that we extend the list of events to include `RND`. It doen't matter where `RND` appears in the list, as long as it is present:

```
eval : Expr -> { [EXCEPTION String, RND, STATE Env] } Eff m Integer

eval (Random upper) = rndInt 0 upper
```

For test purposes, we might also want to print the random number which has been generated:

```
eval (Random upper) = do val <- rndInt 0 upper
                         putStrLn (show val)
                         return val
```

If we try this without extending the effects list, we would see an error something like the following:

```
Expr.idr:28:6:When elaborating right hand side of eval:
Can't solve goal
   SubList [STDIO]
           [(EXCEPTION String), RND, (STATE (List (String, Integer)))]
```

In other words, the `STDIO` effect is not available. We can correct this simply by updating the type of `eval` to include `STDIO`, and setting the context `m` to be one which is handled by `STDIO` such as `IO`:

```
eval : Expr -> { [STDIO, EXCEPTION String, RND, STATE Env] } Eff IO Integer
```

Once effect lists get longer, it can be a good idea instead to encapsulate sets of effects in a type synonym. This is achieved as follows, simply by defining a function which computes a type, since types are first class in IDRIS:

```
EvalEff : Type -> Type
EvalEff t = { [STDIO, EXCEPTION String, RND, STATE Env] } Eff IO t

eval : Expr -> EvalEff Integer
```

4 Dependent Effects

In the programs we have seen so far, the available effects have remained constant. Sometimes, however, an operation can *change* the available effects. The simplest example occurs when we have a state with a dependent type—adding an element to a vector also changes its type, for example, since its length is explicit in the type. In this section, we will see how the `effects` library supports this. Firstly, we will see how states with dependent types can be implemented. Secondly, we will see how the effects can depend on the *result* of an effectful operation. Finally, we will see how this can be used to implement a type-safe and resource-safe protocol for file management.

4.1 Dependent States

Suppose we have a function which reads input from the console, converts it to an integer, and adds it to a list which is stored in a `STATE`. It might look something like the following:

```
readInt : { [STATE (List Int), STDIO] } Eff IO ()
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

But what if, instead of a list of integers, we would like to store a `Vect`, maintaining the length in the type?

```
readInt : { [STATE (Vect n Int), STDIO] } Eff IO ()
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

This will not type check! Although the vector has length `n` on entry to `readInt`, it has length `S n` on exit. The `effects` library allows us to express this as follows:

```
readInt : { [STATE (Vect n Int), STDIO] ==>
            [STATE (Vect (S n) Int), STDIO] } Eff IO ()
readInt = do let x = trim !getStr
             putM (cast x :: !get)
```

The notation `{ xs ==> xs' } Eff m a` in a type means that the operation begins with effects `xs` available, and ends with effects `xs'` available. We have used `putM` to update the state, where the `M` suffix indicates that the *type* is being updated as well as the value. It has the following type:

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

4.2 Result-dependent Effects

Often, whether a state is updated could depend on the success or otherwise of an operation. In our `readInt` example, we might wish to update the vector only if the input is a valid integer (i.e. all digits). As a first attempt, we could try the following, returning a `Bool` which indicates success:

```
readInt : { [STATE (Vect n Int), STDIO] ==>
            [STATE (Vect (S n) Int), STDIO] } Eff IO Bool
readInt = do let x = trim !getStr
             case all isDigit (unpack x) of
                  False => pure False
                  True => do putM (cast x :: !get)
                             pure True
```

Unfortunately, this will not type check because the vector does not get extended in both branches of the `case`!

```
MutState.idr:18:19:When elaborating right hand side of Main.case
block in readInt:
Unifying n and S n would lead to infinite value
```

Clearly, the size of the resulting vector depends on whether or not the value read from the user was valid. We can express this in the type:

```
readInt : { [STATE (Vect n Int), STDIO] ==>
            {ok} if ok then [STATE (Vect (S n) Int), STDIO]
                       else [STATE (Vect n Int), STDIO] } Eff IO Bool
readInt = do let x = trim !getStr
             case all isDigit (unpack x) of
                  False => with_val False (pure ())
                  True => do putM (cast x :: !get)
                             with_val True (pure ())
```

The notation `{ xs ==> res xs' } Eff m a` in a type means that the effects available are updated from `xs` to `xs'`, *and* the resulting effects `xs'` may depend on the result of the operation `res`, of type `a`. Here, the resulting effects are computed from the result `ok`—if `True`, the vector is extended, otherwise it remains the same. We also use `with_val` to return a result:

```
with_val : (val : a) ->
           ({ xs ==> xs' val } Eff m ()) -> { xs ==> xs' } Eff m a
```

We cannot use `pure` here, as before, since `pure` does not allow the returned value to update the effects list. The purpose of `with_val` is to update the effects before returning. As a shorthand, we can write

```
pureM val
```

...instead of...

```
with_val val (pure ())
```

...so our program is:

```
readInt : { [STATE (Vect n Int), STDIO] ==>
            {ok} if ok then [STATE (Vect (S n) Int), STDIO]
                       else [STATE (Vect n Int), STDIO] }
readInt = do let x = trim !getStr
             case all isDigit (unpack x) of
                  False => pureM False
                  True => do putM (cast x :: !get)
                             pureM True
```

When using the function, we will naturally have to check its return value in order to know what the new set of effects is. For example, to read a set number of values into a vector, we could write the following:

```
readN : (n : Nat) ->
        { [STATE (Vect m Int), STDIO] ==>
          [STATE (Vect (n + m) Int), STDIO] } Eff IO ()
readN Z = pure ()
readN {m} (S k) = case !readInt of
                       True => rewrite plusSuccRightSucc k m in readN k
                       False => readN (S k)
```

The `case` analysis on the result of `readInt` means that we know in each branch whether reading the integer succeeded, and therefore how many values still need to be read into the vector. What this means in practice is that the type system has verified that a necessary dynamic check (i.e. whether reading a value succeeded) has indeed been done.

Aside: Only `case` will work here. We cannot use `if/then/else` because the `then` and `else` branches must have the same type. The `case` construct, however, abstracts over the value being inspected in the type of each branch.

4.3 File Management

A practical use for dependent effects is in specifying resource usage protocols and verifying that they are executed correctly. For example, file management follows a resource usage protocol with the following (informally specified) requirements:

- It is necessary to open a file for reading before reading it
- Opening may fail, so the programmer should check whether opening was successful
- A file which is open for reading must not be written to, and vice versa

- When finished, an open file handle should be closed
- When a file is closed, its handle should no longer be used

These requirements can be expressed formally in `effects`, by creating a `FILE_IO` effect parameterised over a file handle state, which is either empty, open for reading, or open for writing. The `FILE_IO` effect's definition is given in Listing 9. Note that this effect is mainly for illustrative purposes—typically we would also like to support random access files and better reporting of error conditions.

Listing 9: File I/O Effect

```
module Effect.File

FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

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

readLine  : Handler FileIO e =>
            { [FILE_IO (OpenFile Read)] } Eff e String
writeLine : Handler FileIO e => String ->
            { [FILE_IO (OpenFile Write)] } Eff e ()
eof       : Handler FileIO e =>
            { [FILE_IO (OpenFile Read)] } Eff e Bool

instance Handler FileIO IO
```

In particular, consider the type of `open`:

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

This returns a `Bool` which indicates whether opening the file was successful. The resulting state depends on whether the operation was successful; if so, we have a file handle open for the stated purpose, and if not, we have no file handle. By `case` analysis on the result, we continue the protocol accordingly.

Listing 10: Reading a File

```
readFile : { [FILE_IO (OpenFile Read)] } Eff IO (List String)
readFile = readAcc [] where
    readAcc : List String -> { [FILE_IO (OpenFile Read)] }
              Eff IO (List String)
    readAcc acc = if (not !eof)
                     then readAcc (!readLine :: acc)
                     else pure (reverse acc)
```

Given a function `readFile` (Listing 10) which reads from an open file until reaching the end, we can write a program which opens a file, reads it, then displays the contents and closes it, as follows, correctly following the protocol:

```
dumpFile : String -> { [FILE_IO (), STDIO] } Eff IO ()
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. For example, changing `open name Read` to `open name Write` yields a compile-time error of the following form:

```
FileTest.idr:16:18:When elaborating right hand side of Main.case
block in testFile:
Can't solve goal
        SubList [(FILE_IO (OpenFile Read))]
                [(FILE_IO (OpenFile Write)), STDIO]
```

In other words: when reading a file, we need a file which is open for reading, but the effect list contains a `FILE_IO` effect carrying a file open for writing.

4.4 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 `open` which did succeed has a corresponding close.

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

```
dumpFile : String -> { [FILE_IO (), STDIO] } Eff IO ()
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 brief clauses for failing matches. Here, for example, we could write:

```
dumpFile : String -> { [FILE_IO (), STDIO] } Eff IO ()
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 `|`. For example, there is a `SYSTEM` effect which supports reading command line arguments, among other things (see Appendix A). To read command line arguments, we can use `getArgs`:

```
getArgs : Handler System e => { [SYSTEM] } Eff e (List String)
```

A main program can read command line arguments as follows, where in the list which is returned, the first element `prog` is the executable name and the second is an expected argument:

```
emain : { [SYSTEM, STDIO] } Eff IO ()
emain = do [prog, arg] <- getArgs
           putStrLn $ "Argument is " ++ arg
           {- ... rest of function ... -}
```

Unfortunately, this will not fail gracefully if no argument is given, or if too many arguments are given. We can use pattern matching bind alternatives to give a better (more informative) error:

```
emain : { [SYSTEM, STDIO] } Eff IO ()
emain = do [prog, arg] <- getArgs | [] => putStrLn "Can't happen!"
                                  | [prog] => putStrLn "No arguments!"
                                  | _ => putStrLn "Too many arguments!"
           putStrLn $ "Argument is " ++ arg
           {- ... rest of function ... -}
```

If `getArgs` does not return something of the form `[prog, arg]` the alternative which does match is executed instead, and that value returned.

5 Creating New Effects

We have now seen several side-effecting operations provided by the `effects` library, and examples of their use in Section 3. We have also seen how operations may *modify* the available effects by changing state in Section 4. We have not, however, yet seen how these operations are implemented. In this section, we describe how a selection of the available effects are implemented, and show how new effectful operations may be provided.

5.1 State

Effects are described by *algebraic data types*, where the constructors describe the operations provided when the effect is available. 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, 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. It is defined as follows in the `effects` library:

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

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
```

Recall that to run an effectful program in `Eff`, we use one of the `run` family of functions to run the program in a particular computation context `m`. For each effect, therefore, we must explain how it is executed in a particular computation context for `run` to work in that context. This is achieved with the following type class:

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

We have already seen some instance declarations in the effect summaries in Section 3. An instance of `Handler e m` means that the effect declared with signature `e` can be run in computation context `m`. The `handle` function takes:

- The `resource` 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.

There are two reasons for taking a continuation here: firstly, this is neater because there are multiple return values (a new resource and the result of the operation); secondly, and more importantly, the continuation can be called zero or more times.

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[4]:

```
using (m : Type -> Type)
  instance Handler State m where
    handle st Get     k = k st st
    handle st (Put n) k = k () n
```

[4]Recall that `using` notation simply gives a type for any implicit arguments in the `using` block

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 m x
get = Get

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

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

An implementation detail (aside): In fact, we are not really using the `Get` and `Put` operations directly. Rather, we are using an `implicit` function which converts an `Effect` to a function in `Eff`, given an automatically constructed proof that the effect is available:

```
implicit
effect' : {a, b: _} -> {e : Effect} ->
          (eff : e t a b) ->
          {prf : EffElem e a xs} ->
          Eff m t xs (\v => updateResTy v xs prf eff)
```

This is the reason for the `Can't solve goal` error when an effect is not available: the implicit proof `prf` has not been solved automatically because the required effect is not in the list of effects `xs`.

Such details are not important for using the library, or even writing new effects, however.

Summary

Listing 11 summarises what is required to define the `STATE` effect.

Listing 11: Complete State Effect Definition

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

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

using (m : Type -> Type)
  instance Handler State m where
     handle st Get     k = k st st
     handle st (Put n) k = k () n

get : { [STATE x] } Eff m x
get = Get

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

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

5.2 Console I/O

Listing 12 gives the definition of the `STDIO` effect, including handlers for `IO` and `IOExcept`. We omit the definition of the top level `Eff` functions, as this merely invoke the effects `PutStr`, `GetStr`, `PutCh` and `GetCh` directly.

Note that in this case, the resource is the unit type in every case, since the handlers merely apply the `IO` equivalents of the effects directly.

Listing 12: Console I/O Effect Definition

```
data StdIO : Effect where
     PutStr : String -> { () } StdIO ()
     GetStr : { () } StdIO String
     PutCh : Char -> { () } StdIO ()
     GetCh : { () } StdIO Char

instance Handler StdIO IO where
    handle () (PutStr s) k = do putStr s; k () ()
    handle () GetStr     k = do x <- getLine; k x ()
    handle () (PutCh c)  k = do putChar c; k () ()
    handle () GetCh      k = do x <- getChar; k x ()

instance Handler StdIO (IOExcept a) where
    handle () (PutStr s) k = do ioe_lift $ putStr s; k () ()
    handle () GetStr     k = do x <- ioe_lift $ getLine; k x ()
    handle () (PutCh c)  k = do ioe_lift $ putChar c; k () ()
    handle () GetCh      k = do x <- ioe_lift $ getChar; k x ()

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

5.3 Exceptions

Listing 13 gives the definition of the `Exception` effect, including two of its handlers for `Maybe` and `List`. The only operation provided is `Raise`.

The key point to note in the definitions of these handlers is that the continuation `k` is not used. Running `Raise` therefore means that computation stops with an error.

5.4 Non-determinism

Listing 14 gives the definition of the `Select` effect for writing non-deterministic programs, including a handler for `List` context which returns all possible successful values, and a handler for `Maybe` context which returns the first successful value.

Here, the continuation is called multiple times in each handler, for each value in the list of possible values. In the `List` handler, we accumulate all successful results, and in the `Maybe` handler we try the first value in the last, and try later values only if that fails.

5.5 File Management

Result-dependent effects are no different from non-dependent effects in the way they are implemented. Listing 15 illustrates this for the `FILE_IO` effect. The syntax for state transitions `{ x ==> {res} x' }`,

Listing 13: Exception Effect Definition

```
data Exception : Type -> Effect where
     Raise : a -> { () } Exception a b

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

instance Handler (Exception a) List where
     handle _ (Raise e) k = []

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

Listing 14: Non-determinism Effect Definition

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

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

instance Handler Selection List where
     handle r (Select xs) k = concatMap (\x => k x r) xs

SELECT : EFFECT
SELECT = MkEff () Selection
```

where the result state `x'` is computed from the result of the operation `res`, follows that for the equivalent `Eff` programs.

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.

6 Example: A "Mystery Word" Guessing Game

In this section, we will use the techniques and specific effects discussed in the tutorial so far to implement a larger example, a simple text-based word-guessing game. In the game, the computer chooses a word, which the player must guess letter by letter. After a limited number of wrong guesses, the player loses[5].

We will implement the game by following these steps:

1. Define the game state, in enough detail to express the rules

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

Listing 15: File I/O Effect Definition

```
data FileIO : Effect where
     Open  : String -> (m : Mode) ->
             {() ==> {res} if res 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

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)

FILE_IO : Type -> EFFECT
FILE_IO t = MkEff t FileIO
```

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

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).

6.1 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
```

6.2 Step 2: Game Rules

We describe the game rules as a 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, and there must be no letters still to be guessed.
- Postcondition: The game is no longer running.

Lost Accept defeat

- Precondition: The game must be running, and there must be 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 more formally 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.

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

6.3 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.

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`:

```
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
```

We leave the definitions of `letters`, `init`, `isElem` and `shrink` as exercises. Having implemented these, the `Handler` implementation for `MysteryRules` is surprisingly straightforward:

```
using (m : Type -> Type)
  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))
```

Each case simply involves directly updating the game state in a way which is consistent with the declared rules. 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.

6.4 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 IO ()
```

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 16.

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 IO ()
runGame = do srand (cast !time)
             let w = index !(rndFin _) words
             NewWord w
             game
             putStrLn !StrState
```

We use the system time (time from the SYSTEM effect; see Appendix A) to initialise the random number generator, then pick a random 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",
         "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.

6.5 Discussion

Writing the rules separately as an effect, then an implementation which uses that effect, ensures that the implementation must follow the rules. This has practical applications in more serious contexts; MysteryRules for example can be though of as describing a *protocol* that a game player most follow, or alternative a *precisely-typed API*.

In practice, we wouldn't really expect to write rules first then implement the game once the rules were complete. Indeed, I didn't do so when constructing this example! Rather, I wrote down a set of likely 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.

Listing 16: Mystery Word Game Implementation

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

- 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.

These are, of course, simple errors, but were caught by the type checker before any testing of the game.

7 Further Reading

This tutorial has given an introduction to writing and reasoning about side-effecting programs in IDRIS, using the `effects` library. More details about the *implementation* of the library, such as how `run` works, how handlers are invoked, etc, are given in a separate paper [2].

Some libraries and programs which use `effects` can be found in the following places:

- `http://github.com/edwinb/SDL-idris` — some bindings for the SDL media library, supporting graphics in particular.
- `http://github.com/edwinb/idris-demos` — various IDRIS demonstration programs, including several `effects` examples from this tutorial, and a "Space Invaders" game.
- `https://github.com/SimonJF/IdrisNet2` — networking and socket libraries.

- `http://github.com/edwinb/Protocols` — a high level communication protocol description language.

The inspiration for the `effects` library was Bauer and Pretnar's Eff language [1], which describes a langauge based on algebraic effects and handlers. Other recent languages and libraries have also been built on this ideas, e.g.[6, 4]. The theoretical foundations are also well-studied [3, 5, 7, 8].

References

[1] A. Bauer and M. Pretnar. Programming with Algebraic Effects and Handlers, 2012. Available from `http://arxiv.org/abs/1203.1539`.

[2] E. Brady. Programming and reasoning with algebraic effects and dependent types. In G. Morrisett and T. Uustalu, editors, *International Conference on Functional Programming (ICFP '13)*, pages 133–144. ACM, 2013.

[3] M. Hyland, G. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357:70–99, 2006.

[4] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th International Conference on Functional Programming (ICFP '13)*. ACM, 2013.

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

[6] 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.

[7] 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.

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

[9] The Idris Community. Programming in Idris : a tutorial. `http://idris-lang.org/tutorial`, 2014.

A Effects Summary

This appendix gives interfaces for the core effects provided by the `effects` library.

A.1 EXCEPTION

```
module Effect.Exception

EXCEPTION : Type -> EFFECT

raise : a -> { [EXCEPTION a ] } Eff m b

instance           Handler (Exception a) Maybe
instance           Handler (Exception a) List
instance           Handler (Exception a) (Either a)
instance           Handler (Exception a) (IOExcept a)
instance Show a => Handler (Exception a) IO
```

A.2 FILE_IO

```
module Effect.File

FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

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

readLine  : Handler FileIO e =>
            { [FILE_IO (OpenFile Read)] } Eff e String
writeLine : Handler FileIO e => String ->
            { [FILE_IO (OpenFile Write)] } Eff e ()
eof       : Handler FileIO e =>
            { [FILE_IO (OpenFile Read)] } Eff e Bool

instance Handler FileIO IO
```

A.3 RND

```
module Effect.Random

RND : EFFECT

srand  : Integer ->             { [RND] } Eff m ()
rndInt : Integer -> Integer -> { [RND] } Eff m Integer
rndFin : (k : Nat) ->           { [RND] } Eff m (Fin (S k))

instance Handler Random m
```

A.4 SELECT

```
import Effect.Select

SELECT : EFFECT

select : List a -> { [SELECT] } Eff m a

instance Handler Selection Maybe
instance Handler Selection List
```

A.5 STATE

```
module Effect.State

STATE : Type -> EFFECT
```

```
get    :             { [STATE x] } Eff m x
put    : x ->        { [STATE x] } Eff m ()
putM   : y ->        { [STATE x] ==> [STATE y] } Eff m ()
update : (x -> x) -> { [STATE x] } Eff m ()

instance Handler State m
```

A.6 STDIO

```
module Effect.StdIO

STDIO : EFFECT

putChar  : Handler StdIO m => Char ->   { [STDIO] } Eff m ()
putStr   : Handler StdIO m => String -> { [STDIO] } Eff m ()
putStrLn : Handler StdIO m => String -> { [STDIO] } Eff m ()

getStr   : Handler StdIO m =>           { [STDIO] } Eff m String
getChar  : Handler StdIO m =>           { [STDIO] } Eff m Char

instance Handler StdIO IO
instance Handler StdIO (IOExcept a)
```

A.7 SYSTEM

```
module Effect.System

SYSTEM : EFFECT

getArgs : Handler System e =>           { [SYSTEM] } Eff e (List String)
time    : Handler System e =>           { [SYSTEM] } Eff e Int
getEnv  : Handler System e => String -> { [SYSTEM] } Eff e (Maybe String)

instance Handler System IO
instance Handler System (IOExcept a)
```