Mathematically Structuring Programming Languages

Cambridge Programming Research Group Lectures, May 2010

Dominic Orchard[1]

1 Introduction

Defining the semantics of a programming language is a mathematical task that is made succinct and general by the use of abstract mathematical structures. Furthermore, programming can be made succinct and general by using abstract mathematical structures to abstract and restructure parts of a program. In this lecture, we intersect both activities, of programming and defining semantics, using the functional language Haskell as a meta language to describe a simple arithmetic language E. Using Haskell as a meta language gives us a semantical definition of E and an interpreter for free, where E is a *shallow embedded* DSL (see Lecture 2). We then extend the language E with *exceptions*, a kind of computational *effect*, and employ the category theoretic notion of a *monad* to abstract the handling of exceptions, rendering our semantics clean and concise. Monads are a useful tool to have in our programming/semantics toolbox.

Knowledge of Haskell is not necessary, although might help. Up-to-date notes, slides, errata, and examples can be found on the author's homepage[1].

1.1 Objectives

After completing this lecture students should:

- have gained a feel for using a language as a meta language to describe another,
- understand the concept of a monad and the use of monads for structuring effects, in particular exceptions,
- understand the merit of using abstract mathematical structures to help structure programs and language semantics.

[1]Homepage: `http://www.cl.cam.ac.uk/~dao29`

2 Meta Programming a Simple Language

2.1 Simple arithmetic language *E*

Consider the following simple, prefix, arithmetic language with expressions E defined by the grammar:

$$E := \texttt{add}\ E_1\ E_2 \mid \texttt{sub}\ E_1\ E_2 \mid \texttt{div}\ E_1\ E_2 \mid \texttt{mul}\ E_1\ E_2 \mid n$$

The denotational semantics of E could be defined by the following function $[\![]\!]$ mapping the syntactic category of expressions E to some domain D:

$$\begin{aligned}
[\![\_]\!] &: E \to D \\
[\![\texttt{add}\ e1\ e2]\!] &= [\![e1]\!] + [\![e2]\!] \\
[\![\texttt{sub}\ e1\ e2]\!] &= [\![e1]\!] - [\![e2]\!] \\
[\![\texttt{div}\ e1\ e2]\!] &= \frac{[\![e1]\!]}{[\![e2]\!]} \\
[\![\texttt{mul}\ e1\ e2]\!] &= [\![e1]\!] * [\![e2]\!] \\
[\![n]\!] &= n
\end{aligned}$$

On the right hand side of $[\![]\!]$ we give the semantics of E-expressions in terms of another language on the domain D. Here we presume standard mathematical definitions of integer arithmetic, but there is one caveat which is often a point of contention: for any value of x, what should the behaviour of $\frac{x}{0}$ be?

2.2 Defining an executable semantics

We describe the semantics of E in domain D using a *meta-language*, and if such a language happens to be an executable programming language then defining $[\![]\!]$ gives us an executable semantics, and thus an interpreter for free. Functional programming languages, such as ML and Haskell, are suitable candidates for such a meta-language as they tend to have clean, lightweight syntax and semantics that are similar to that of functions and sets in the mathematical sense. The additional features of pattern matching makes the business of dealing with syntax trees much more straightforward; the above mathematical definition is not far off ML or Haskell syntax.

For the rest of this lecture we will assume that our meta-language is Haskell, but ML would be just as suitable. If you are more familiar with ML but not Haskell, Appendix A provides a few brief pointers.

As our grammar is suitably simple, where productions fit an applicative-style: $e_1\ e_2$, we can encode the language as a *shallow* embedding in Haskell, using function definitions for each non-terminal of our grammar [1]:

[1] All code samples can be downloaded from the author's homepage: `http://www.cl.cam.ac.uk/~dao29/cprg-lecture.html`

```
add x y = x + y
sub x y = x - y
div x y = x `Prelude.div` y
mul x y = x * y
```

We can the write programs in Haskell in the E sub-language, and evaluate E programs to Haskell values which could be passed to further Haskell code. The definition also provides a typed version of E for free, as we reuse Haskell's type inference and type checking by shallow embedding the language as functions. As such, only well-typed E are well-formed, e.g:

```
add (mul 4) 5
```

is not well-formed, and is not accepted as it raises a type error: `Couldnt match expected type 'Int' against inferred type 'Int -> Int'`.

Haskell infers some fairly general types for our operations, such as:

$$\texttt{add} :: \mathit{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$$

This type comes from the type of $+$ in Haskell which is overloaded for any types which are a member of the *Num* class, i.e. the types for which $+$ has been defined (for more on Haskell type classes see `http://www.haskell.org/tutorial/classes.html`). For our language we constrain the operations to be just those on the built-in type *Int* in Haskell. We give our first definition of E in **Figure 1**. Compare the definition to that of $[\![]\!]$ above and note how succinct and clean the definition is. In defining semantics we would like to keep our definitions fairly succinct, and as we add features to the language we would like the semantics of the new features to be as decoupled from the existing semantics as possible, even if the new features are not completely orthogonal.

```
add :: Int -> Int -> Int
add x y = x + y

sub :: Int -> Int -> Int
sub x y = x - y

div :: Int -> Int -> Int
div x y = x `Prelude.div` y

mul :: Int -> Int -> Int
mul x y = x * y
```

Figure 1: Semantics of E

Example The following expressions:

```
x = add 1 2
y = add (mul 3 4) 5
z = div 1 0
```

evaluate to integer results where $x = 3$ and $y = 17$, except z which does not evaluate to a value but instead causes a Haskell exception: `*** Exception: divide by zero`.

2.3 Adding exceptions

Instead of raising an exception in our meta language we want to handle and output exceptions within the language E. We can encode the fact that `div` possibly returns an exception by returning a sum type, such as

$$\mathbb{Z} + e$$

where e is the type of an exception, or error. In Haskell we can construct arbitrary sum types using algebraic datatypes, but we also have the built-in *Either* type, which provides a binary sum type:

$$\mathit{Either}\ a\ b = a + b$$

with injections to this type via the value constructors *Left* and *Right*:

$$\begin{aligned} \mathit{Left} &:: a \rightarrow \mathit{Either}\ a\ b \\ \mathit{Right} &:: b \rightarrow \mathit{Either}\ a\ b \end{aligned}$$

We redefine `div` to return a string with an error message if the denominator is 0, else return the integer result of the division:

```
div :: Int -> Int -> Either Int String
div x y = if y==0 then Right "Divide by zero"
                     else Left (x ‘Prelude.div‘ y)
```

Example Evaluating the following expressions:

```
x = add 1 2
y = add (mul 3 4) 5
z = div 1 0
```

gives $x = 3$, $y = 17$, and $z = \mathit{Right}$ “*Divide by zero*”. What does `w` evaluate to in the following definition?

```
w = add 1 (div 1 0)
```

The right hand side of `w` is not well-typed and will not be accepted by Haskell because `div` returns a value of type *Either Int String* while the second parameter of `add` should be of type *Int*, thus a type mismatch occurs.

In order to propagate our exceptions through the expression language we need to accept possible exceptions as parameters to any of the operations and to thread possible exceptions through. We give the definition of E with this extension in **Figure 2**.

```
add :: Either Int String -> Either Int String -> Either Int String
add x y = case x of
            Right e -> Right e
            Left x' -> case y of
                          Right e' -> Right e'
                          Left y'  -> Left (x' + y')

sub :: Either Int String -> Either Int String -> Either Int String
sub x y = case x of
            Right e -> Right e
            Left x' -> case y of
                          Right e' -> Right e'
                          Left y'  -> Left (x' - y')

div :: Either Int String -> Either Int String -> Either Int String
div x y = case x of
            Right e -> Right e
            Left x' -> case y of
                          Right e' -> Right e'
                          Left y'  -> if y'==0 then
                                          Right "Divide by zero"
                                               else
                                          Left (x' `Prelude.div` y')

mul :: Either Int String -> Either Int String -> Either Int String
mul x y = case x of
            Right e -> Right e
            Left x' -> case y of
                          Right e' -> Right e'
                          Left y'  -> Left (x' * y')
```

Figure 2: Semantics of E with exceptions

We can now thread exceptions through expressions, but we have to lift integers to the *Either* type using the *Left* injection.

Example The following expressions:

```
x = add (Left 1) (Left 2)
y = add (mul (Left 3) (Left 4)) (Left 5)
```

```
z = div (Left 1) (Left 0)
w = add (Left 1) (div (Left 1) (Left 0))
```

evaluate to $x = \mathit{Left}\ 3$, $y = \mathit{Left}\ 17$, and both z and w evaluate to *Right* "*Exception: Divide by zero*".

2.4 Adding exceptions with abstraction

We have now added exceptions, but at the cost of making our semantics significantly more complicated and hard to understand. The code for handling exceptions is the same in each definition, apart from the inner case where a value is computed, thus we can abstract the exception handling code to try and make our semantics succinct once again. We could use macros as discussed in Lecture 2, but in functional languages we can use *higher-order* functions to much the same effect because we don't need to deconstruct the parameter of the "macro". We define such a "macro" as the higher-order function `handle` which takes a function over which to handle exceptions. **Figure 3** gives the semantics. (Note: we now use the *point-free*, or *pointless*, style of function definition for `add` etc.)

```
add :: Either Int String -> Either Int String -> Either Int String
add = handle (\x y -> Left (x + y))

sub :: Either Int String -> Either Int String -> Either Int String
sub = handle (\x y -> Left (x - y))

div :: Either Int String -> Either Int String -> Either Int String
div = handle (\x y -> if y==0 then Right "Exception: Divide by zero"
                              else Left (x ‘Prelude.div‘ y))

mul :: Either Int String -> Either Int String -> Either Int String
mul = handle (\x y -> Left (x * y))

handle :: (a -> a -> Either String a) -> Either String a -> Either String a
          -> Either String a
handle f x y = case x of
                Right e -> Right e
                Left x' -> case y of
                             Right e' -> Right e'
                             Left y'  -> f x' y'
```

Figure 3: Semantics of E with abstracted exception handling

Example The following expressions:

```
x = add (Left 1) (Left 2)
y = add (mul (Left 3) (Left 4)) (Left 5)
```

```
z = div (Left 1) (Left 0)
w = add (Left 1) (div (Left 1) (Left 0))
```

evaluate to $x = \textit{Left}\ 3$, $y = \textit{Left}\ 17, z = \textit{Right}\ ``\textit{Exception: Divide by zero}''$, and $w = \textit{Right}\ ``\textit{Exception: Divide by zero}''$.

2.5 Further abstraction

We must still inject values going into our language, but we have a much shorter definition of the semantics now, with the `handle` performing the exception threading for us. Yet, there is still some code duplication: we have to check both arguments for exceptions, with essentially the same code. Can we do better? Consider the following function `extend` which is similar to the `handle` function but only handles unary functions as opposed to binary:

```
extend :: (a -> Either a e) -> Either a e -> Either a e
extend f x = case x of Right e -> Right e
                       Left x  -> f x
```

We can then rewrite the binary `handle` function from before in terms of the unary `extend` function:

```
handle f x y = extend (\x' -> extend (\y' -> f x' y') y) x
```

Instead of keeping all the definitions of `add`, `mul`, etc. in terms of `handle`, we now express all operations in a simpler form where we only consider exceptions in the return value, and not on any of the inputs:

```
add :: Int -> Int -> Either Int String
add x y = Left (x + y)
```

We can apply `add` to simple integers, or if we are applying it to the result of another computation which may have returned an exception then we can lift the parameters using `extend`, thus we move the handling of exceptions outside of the function. E.g.

```
w = extend (\a -> add 1 a) (div 1 0)
```

The first argument to `extend` has correct type: ($\textit{Int} \rightarrow \textit{Either Int String}$). If the second argument is an exception value (*Right* value) then *extend* passes on the exception without even calling the function (`\a -> add 1 a`), or if the second argument is a *Left* value, then *extend* unwraps it from the *Left* constructor and passes the value to the first argument function.

Figure 4 gives the full semantics using `extend`, with extra functions `return` (of type $a \to Either\ a\ b$) and `exception` (of type $b \to Either\ a\ b$) wrapping the *Left* and *Right* constructors making it easier to read and understand the intention of the `add`, `sub`, `div`, and `mul` definitions.

```
return x = Left x
exception x = Right x

extend :: (a -> Either a e) -> Either a e -> Either a e
extend f x = case x of Right e -> Right e
                       Left x  -> f x

add :: Int -> Int -> Either Int String
add x y = return (x + y)

sub :: Int -> Int -> Either Int String
sub x y = return (x - y)

div :: Int -> Int -> Either Int String
div x y = if y==0 then exception "Exception: Divide by zero"
                  else return (x ‘Prelude.div‘ y)

mul :: Int -> Int -> Either Int String
mul x y = return (x * y)
```

Figure 4: Semantics of E with further abstracted exception handling

Example The following expressions

```
x = add 1 2
y = extend (\a -> add a 5) (mul 3 4)
z = div 1 0
w = extend (\a -> add 1 a) (div 1 0)
v = extend (\a -> extend (\b -> mul a b) (add 2 4)) (add 4 3)
```

evaluate to $x = Left\,3$, $y = Left\,17$, $z = Right$ “*Exception: Divide by zero*″, and $w = Right$ “*Exception: Divide by zero*″ as before, and $v = Left\,42$, showing how `extend` can be used for handling exceptions on both arguments of an operator.

The definition of our semantics for E with exception in **Figure 4** is now considerably simpler and easier to read than our previous attempts in **Figure 2** and **Figure 3**. However, writing expressions in E is now much more cumbersome, with lambda abstractions and calls to `extend` everywhere. We will improve this ugliness later.

The definition in **Figure 4** actually uses a *monad* structure to define the handling of exceptions; we now introduce *monads*.

3 Monads

Whilst we have been abstracting the semantics of our language we have essentially defined the operations of a structure called a *monad.* The concept of a monad comes from a branch of abstract mathematics called *category theory.* Category theory essentially provides a framework for talking about relationships, such as equality, between mathematical structures, such as groups and sets, but at a higher level of abstraction. Since the early '90s, computer scientists have realised that structures in category theory give succinct ways to describe certain types of computation, and whose laws and operations correspond to many features in our languages. As such, these structures can an extremely useful tools for structuring and reasoning in programming and theoretical programming language research. We will not discuss the category theoretic understanding of monads (although a brief introduction is given in Appendix B) but will instead give a functional programming interpretation. For further reading see [1, 2].

Monads provide a succinct way to structure computations which have some kind of *effect.* A function with an effect can be thought of as a normal function along with some extra information contained, or associated, with its return value e.g. $f' : a \to M\, b$ is a function $a \to b$ with a wrapper M containing information about some effects along with the return value b.

In our example language, the effect is exceptions, but there are many other types of effect which can be similarly structured by monads such as state, input/output, and non-determinism.

Monads will give us a tool in our toolbox of programming/semantics tricks for handling effects. As a warm-up exercise to thinking about mathematical structures we will first recall a more well-known example: *monoids*

Definition A monoid is a triple $(S, \bullet, e)$ of a set S, a binary operation $\bullet : S \times S \to S$, and an element $e : S$ with the following axioms:

- [identity] $\forall x \,.\, x \bullet e = x = e \bullet x$
- [associativity] $\forall xyz \,.\, x \bullet (y \bullet z) = (x \bullet y) \bullet z$

The following are example monoids:

- $(\mathbb{N}, +, 0)$
- $(\mathbb{N}, \times, 1)$
- $([a], +\!+, [])$

The following are not monoids as the associativity axiom does not hold:

- $(\mathbb{N}, -, 0)$

- $(\mathbb{Z}, \backslash, 1)$

Definition A monad M is a *functor*, which means, in the functional programming interpretation, a monad is a data structure M, parameterised by an element type a along with an operation:

$$M_{map} : (a \to b) \to M\ a \to M\ b$$

which lifts a function to operate on the elements of the monad (cf. map on lists – lists are functors with the normal list map function). The M_{map} function preserves the monad's information/structure from the argument in the return monad. We could think of a monad as a kind of container for values of type a. For the M functor to be a monad it must be equipped with the following two polymorphic functions:

- $\eta : a \to M\,a$
- $\mu : M(M\,a) \to M\,a$

Where η is often called *unit* or *return* in FP and μ is often often called *join* in FP.

A monad is a bit like a *monoid*, but instead of operating on a set S, we are operating on $M\,a$ values, where:

- η constructs an identity element
- μ is the binary operation but instead of taking a pair of values like the monoid $\bullet : S \times S \to S$ the pair is now compositions of M, $\mu : M(M\,a) \to M\,a$

As such, we have similar identity and associativity laws, although they look slightly different and are a bit harder to understand so we will not delve into them here (see Appendix B).

If we want to use monads to structure our computations in E then we need to be able to compose functions with types like $a \to M\,b$. As we saw earlier, functions of this type do not compose well, e.g. if we have $f : a \to M\,b$ and $g : b \to M\,c$, we get a type mismatch if we try to do $g\,(f\,x)$ as the return type of f does not match the parameter type of g.

The operations of our monad give us a way to construct such a composition by turning $g : b \to M\,c$ into the function $g' : M\,b \to M\,c$. An operation called *extend* provides this construction and is derived from the monad[2].

$$\begin{aligned} \mathit{extend} &: (a \to M\ b) \to M\ a \to M\ b \\ \mathit{extend}\, f &= \mu \circ (M_{map}\, f) \end{aligned}$$

[2]See Appendix B for more details of this construction which comes from a *Kleisli* category of a monad.

e.g. when applied to g:

$$\begin{aligned} M_{map}\, g &: M\, b \rightarrow M(M\, b) \\ \mu \circ (M_{map}\, g) &: M\, b \rightarrow M\, c \end{aligned}$$

Thus we can compose f and g by writing:

$$\begin{aligned} & \mu \circ (M_{map}\, g) \circ f && : a \rightarrow M\, c \\ \Rightarrow \quad & (\mathit{extend}\, g) \circ f && : a \rightarrow M\, c \end{aligned}$$

The *extend* operation essentially lifts a function of type $(a \rightarrow M\ b)$ to be a function of type $(M\ a \rightarrow M\ b)$, over the monad. The type of the first argument to *extend* means that we do not have to deconstruct or understand the monad structure M in our functions – we can only construct effects, not take them apart. The *extend* function looks like it deconstructs the monad for us, giving us a value a out of the monad M. In fact, as we can see from the derivation above, *extend* actually uses M_{map} to lift our function to one that takes a monad $M\, a$ and returns a new monad that wraps the old monad $M(M\, b)$ which *join* combines together for us. Thus incoming effects are combined with outgoing effects.

The *extend* operation also has its own laws for preserving identity and associativity:

- {Left identity}
  $\mathit{extend}\, f\ (\mathit{unit}\, x) \equiv f x$
- {Right identity}
  $\mathit{extend}\ \mathit{unit}\, x \equiv x$
- {Associativity}
  $\mathit{extend}\, g\ (\mathit{extend}\, f\ x) \equiv \mathit{extend}\, (\lambda x'\ .\ \mathit{extend}\, g\ (f x'))\ x$

In our semantics for E we have been using the monad of a sum type – *Either* in this case. We have already defined the *unit* (which we called `return`) and *extend* function in the **Figure 4**. Check their signatures to see that they match the signatures of *unit* and *extend* above. Thus, we are using *extend* in a form not derived from the underlying M_{map} and *join* operations. To be sure we have really defined correct operations of a monad, and that the sum types can really be a monad, we must verify the axioms of identity and associativity. Here we verify the laws in terms of *extend* as these laws are slightly easier to understand, but we can do the same for the monad identity and associativity laws.

Theorem 3.1 *A sum type* $F\ a\ b = a + b$ *is a monad.*

Proof Take the operations *unit* and *extend* to be those defined in **Figure 4**, using two definitions for *extend* in a pattern matching style instead of a `case` expression, and injections $inl : \forall ab \,.\, a \to a + b$ and $inr : \forall ab \,.\, b \to a + b$ (equivalent to *Left* and *Right* constructors in Haskell):

$$\begin{aligned} &unit = inl \\ &extend\ f\ (inl\ x) = f x \\ &extend\ f\ (inr\ e) = inr\ e \end{aligned}$$

- Proof of left identity: $extend\ f\ (unit\ x) \equiv f x$

  $$\begin{array}{lll} & extend\ f\ (unit\ x) & \\ \Leftrightarrow & extend\ f\ (inl\ x) & \{\text{definition of } unit\} \\ \Leftrightarrow & f x & \{\text{definition of } extend \text{ - case 1}\} \end{array}$$

  The sum-monad satisfies the left identity property. □

- Proof of right identity: $extend\ unit\ x \equiv x$

  x may be of form $(inl\ x')$ or $(inr\ x')$ thus we consider both cases.

  $$\begin{array}{lll} (1) & extend\ unit\ (inl\ x') & \\ \Leftrightarrow & unit\ x' & \{\text{definition of } extend\} \\ \Leftrightarrow & inl\ x' & \{\text{definition of } unit\} \\ & & \\ (2) & extend\ unit\ (inr\ x') & \\ \Leftrightarrow & inr\ x' & \{\text{definition of } extend\} \\ & & \\ \therefore & extend\ unit\ x & \\ \Leftrightarrow & x & \{(1) \text{ and } (2)\} \end{array}$$

  The sum-monad satisfies the right identity property. □

- Proof of associativity: left as an exercise (hint: express *extend* with `case` expressions instead of pattern matching as in **Figure 4**, then expand out the definitions, and use the associativity of nested `case` expressions:

$$\begin{array}{lcl} \texttt{case}\ (\texttt{case}\ e\ \texttt{of}\ b_1 \to e_1\ \ldots)\ \texttt{of} & & \texttt{case}\ e\ \texttt{of} \\ \quad b'_1 \to e'_1 & \Leftrightarrow & \quad b_1 \to (\texttt{case}\ e_1\ \texttt{of}\ b'_1 \to e'_1 \ldots) \\ \quad \ldots & & \quad \ldots \end{array}$$

The sum-type $\forall ab \,.\, a + b$ is a monad. □.

3.1 Monads in Haskell

As monads structure effects, Haskell provides an embedded imperative language in its syntax, which is parameterisable by a monad describing the way that effects are handled through the computation. This syntax considerably improves programming with monads, as expressions such as:

```
v = extend (\a -> extend (\b -> mul a b) (add 2 4)) (add 4 3)})
```

can be difficult to read and write. The above example written in the monadic *do* notation, as it is called, becomes the more readable and writable:

```
v = do a <- add 4 3
       b <- add 2 4
       mul a b
```

The binding construction: `<-` takes a computation and binds it to a variable, which is accessible by future computations with effects handled by the *extend* operation. The syntax:

```
do y <- f x
   g y
```

is desugarded into:

```
extend (\y -> g y) (f x)
```

In Haskell, the *extend* operation is the (`>>=`) operator, called *bind*, which has its arguments reversed, thus has type:

$$\texttt{>>=} : M\ a \to (a \to M\ b) \to M\ b$$

as opposed to our *extend* function which had type:

$$extend : (a \to M\ b) \to M\ a \to M\ b$$

This operator, (`>>=`), can be used infix, conveying the intuition that *extend* pushes a monadic value into a computation that takes a pure argument e.g.

```
  extend (\a -> mul 3 a) (add 2 4)
≡ (add 2 4) >>= (\a -> mul 3 a)
```

In the *do*-notation the monad laws make even more intuitive sense:

- {Left identity}

```
do x' <- return x
   f x'            =   do f x
```

- {Right identity}

```
do x <- m
   return x        =   do m
```

- {Associativity}

```
do  y <- do { x <- m  =  do  x <- m
              f x            do  { y <- f x
            }                      g y
    g y                          }
                      =  do  x <- m
                             y <- f x
                             g y
```

3.2 E with Haskell monads and *do* notation

Haskell defines a monad *type class*, providing a way to overload the built-in `>>=` and `return` monad operations for some type that they are not currently defined for. Haskell's type class mechanism allows overloading in the language, where a function can have multiple definitions indexed by the types of their parameters. The *do* notation is implicitly parameterised by a monad whose operations are chosen based on the types of the parameters. We implement an instance of Haskell's monad type class to give our final semantics, which is not only succinct in its definition because of the use of monads, but which let's us write E expressions succinctly using the *do* notation.

Figure 5 gives the code for the executable semantic definition. We use the custom data type `Value` for clarity, which is essentially the `Either` type, but with more appropriate names than *Left* and *Right*.

Example The following expressions:

```
x = add 1 2
y = do a <- mul 3 4
       add a 5
z = div 1 0
w = do a <- div 1 0
       add 1 a
v = do a <- add 4 3
       b <- add 2 4
       mul a b
```

evaluate to $x = \textit{Value}\ 3$ and $y = \textit{Value}\ 17$, and both z and w evaluate to *Exception "Divide by zero"*, and $v = \textit{Value}\ 42$.

```
data Value a = Value a | Exception String deriving Show

instance Monad Value where
    return x = Value x
    x >>= f = case x of Exception e -> Exception e
                        Value x     -> f x

add :: Int -> Int -> Value Int
add x y = return (x + y)

sub :: Int -> Int -> Value Int
sub x y = return (x - y)

div :: Int -> Int -> Value Int
div x y = if y==0 then Exception "Divide by zero"
                  else return (x `Prelude.div` y)

mul :: Int -> Int -> Value Int
mul x y = return (x * y)
```

Figure 5: Semantics of E with exceptions using Haskell monads

4 Concluding Remarks

Monads provide a way to structure computations with some kind of *effect*. They provide modularity, allowing us to describe simpler computations without repeating code for threading or combining effects; they provide isolation for effects, and give a clear definition of how these effects are sequenced.

A particular useful monad is the IO monad which structures input/output effects, describing how these effects are sequenced thus giving a clear semantics for the ordering of input/output. e.g.

```
do  putStr "Enter a number:"
    x <- getLine
    double_x <- return ((read x) * 2)
    putStr ("Double your number = " ++ (show double_x))
```

The two functions *putStr* and *getLine* have types:

$$putStr :: String \rightarrow \texttt{IO}\ ()$$
$$getLine :: \texttt{IO}\ String$$

thus produce effects, which the *extend*/*bind*, combines together, sequencing the effects in the correct order, which is especially vital for input/output, but which might not normally be provided by a language with lazy evaluation such as Haskell. Note the use of `return` in the third line to inject the pure computation `((read x) * 2)` into the IO monad, producing an empty IO effect.

The IO monad is a special case of the more general *state* monad, which describes state-changing effects. The monad is of type:

$$\mathit{State}\ a = s \rightarrow s \times a$$

where s is the type of some state value. Thus a $\mathit{State}\ a$ value takes a state of type s and return s anew state (which may be an altered version of the argument state) along with a result value. Reading and writing to memory is another effect that can be structured using a state monad. Other monads include non-determinism, choice, and continuations. For further reading see [1, 2].

In this lecture we have introduced a small arithmetic language E, and have described its semantics using Haskell as a meta-language, gaining an interpreter for free. The definition of E was given in a *shallow* style: essentially a syntax-directed approach to semantics where we gave definitions by way of Haskell function definitions for each terminal in the language, thus providing a shallow embedded DSL. We added exceptions to the language using the abstraction of a *monad*, succinctly describing the handling of exceptions in E. We then reused Haskell's monadic *do* notation to hide the operations of the monad in E expressions.

In these notes we arrived at describing exceptions with a monad ourselves by trying to abstract and reuse code. Now that we are equipped with the knowledge that monads structure effects we can jump straight to using a monad anytime we realise we are dealing with effectful computations in a program, saving time and improving the quality of our code.

References

[1] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[2] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

A Notes on Haskell for ML Programmers

A few quick notes on some syntactic differences between ML and Haskell. While ML expresses anonymous functions (lambda abstractions) with the syntax:

```
(fn x => x)
```

Haskell syntax is:

```
(\x -> x)
```

The parametrically polymorphic list type in ML may be written:

```
'a list
```

whilst in Haskell is is written:

```
[a]
```

In general, type parameters are written in postfix style in Haskell, whereas they are prefix in ML, e.g. `'a 'b Either` in ML vs. `Either a b` in Haskell.

B Monads: Categorically

Definition Briefly, a *monad* is a triple of an endofunctor $M : \mathcal{C} \to \mathcal{C}$ and two natural transformations: (M, η, μ) where:

[M1] $\eta : 1_{\mathcal{C}} \Rightarrow M$

[M2] $\mu : M^2 \Rightarrow M$

[M3] Coherence condition:

- $\mu \circ M\mu = \mu \circ \mu M$ (akin to associativity)
- $\mu \circ M\eta = \mu \circ \eta M = 1_M$ (akin to identity)

Definition Given a monad M for a category $\mathcal{C}$, a *Kleisli* category $\mathcal{C}_M$ is a category whose objects are the objects of $\mathcal{C}$ and whose morphisms $f_M : A \to B$ are the morphisms $f : A \to MB$ in $\mathcal{C}$. Identity in the $\mathcal{C}_M$ is arises from the η natural transformation, and composition $f_M : A \to B$, $g_M : B \to C$, $g \circ_M f : A \to C$ composes morphisms in $\mathcal{C}$ of $f : A \to MB$, $g :: B \to MC$ to $g \circ f : A \to MC$.

Extension within the Kleisli category is provided by an operator *extend* which takes a Kleisli morphism in $\mathcal{C}_M$ and extends the method in the underlying category $\mathcal{C}$ from a monad to a monad i.e.

$$\mathit{extend} : \forall a, b \;.\; (a \to M\ b) \to (M\ a \to M\ b)$$

The *extend* operation can be derived from the monad's functor M and μ as such:

$$\mathit{extend}\, f = \mu \circ M\, f$$

The application of the functor M to f has type $M\, f : M\, a \to M(M\, b)$ creating a nested monad over b which μ combines to $M\, b$.

The *extend* operation of a monad can be used to define composition in the *Kleisli* category:

$$\forall x \;.\; (g \circ_M f)\ x \equiv \mathit{extend}\, g\ (fx)$$

A *Kleisli* category provides a framework for working with monads. Using monadic *do* notation in Haskell is like working in a Kleisli category.