Algebraic structures

Lecture 18 Thursday, March 30, 2023

In abstract algebra, algebraic structures are defined by a set of elements and operations on those elements that satisfy certain laws. Some of these algebraic structures have interesting and useful computational interpretations. In this lecture we will consider several algebraic structures (monoids, functors, and monads), and consider the computational patterns that these algebraic structures capture. We will look at Haskell, a functional programming language named after Haskell Curry, which provides support for defining and using such algebraic structures. Indeed, monads are central to practical programming in Haskell. First, however, we consider type constructors, and see two new type constructors.

1 Type constructors

A *type constructor* allows us to create new types from existing types. We have already seen several different type constructors, including product types, sum types, reference types, and parametric types.

The product type constructor $\times$ takes existing types τ_1 and τ_2 and constructs the product type $\tau_1 \times \tau_2$ from them. Similarly, the sum type constructor $+$ takes existing types τ_1 and τ_2 and constructs the sum type $\tau_1 + \tau_2$ from them.

We will briefly introduce list types and option types as more examples of type constructors.

1.1 Lists

A list type τ **list** is the type of lists with elements of type τ. We write $[\,]$ for the empty list, and $v_1 :: v_2$ for the list that contains value v_1 as the first element, and v_2 is the rest of the list. We also provide a way to check whether a list is empty (isempty? e) and to get the head and the tail of a list (head e and tail e).

Assume that we have a call-by-value lambda calculus with booleans and a fixpoint operator $\mu x : \tau.\ e$. We extend the syntax and semantics of this language with lists as follows.

Expressions	$e ::= \cdots \mid [\,] \mid e_1 :: e_2 \mid \mathsf{isempty?}\ e \mid \mathsf{head}\ e \mid \mathsf{tail}\ e$
Values	$v ::= \cdots \mid [\,] \mid v_1 :: v_2$
Types	$\tau ::= \cdots \mid \tau\ \mathbf{list}$
Evaluation contexts	$E ::= \cdots \mid E :: e \mid v :: E \mid \mathsf{isempty?}\ E \mid \mathsf{head}\ E \mid \mathsf{tail}\ E$

$$\frac{}{\mathsf{isempty?}\ [\,] \longrightarrow \mathbf{true}} \qquad \frac{}{\mathsf{isempty?}\ v_1 :: v_2 \longrightarrow \mathbf{false}} \qquad \frac{}{\mathsf{head}\ v_1 :: v_2 \longrightarrow v_1} \qquad \frac{}{\mathsf{tail}\ v_1 :: v_2 \longrightarrow v_2}$$

$$\frac{}{\Gamma \vdash [\,] : \tau\ \mathbf{list}} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\ \mathbf{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \mathbf{list}} \qquad \frac{\Gamma \vdash e : \tau\ \mathbf{list}}{\Gamma \vdash \mathsf{isempty?}\ e : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : \tau\ \mathbf{list}}{\Gamma \vdash \mathsf{head}\ e : \tau} \qquad \frac{\Gamma \vdash e : \tau\ \mathbf{list}}{\Gamma \vdash \mathsf{tail}\ e : \tau\ \mathbf{list}}$$

For example, we can define a function append that takes two lists and appends one to the other, as follows.

$$\mathsf{append} \triangleq \mu f : \tau\ \mathbf{list} \to \tau\ \mathbf{list} \to \tau\ \mathbf{list}.\ \lambda a : \tau\ \mathbf{list}.\ \lambda b : \tau\ \mathbf{list}.\ \mathbf{if}\ \mathsf{isempty?}\ a\ \mathbf{then}\ b\ \mathbf{else}\ (\mathsf{head}\ a) :: (f\ (\mathsf{tail}\ a)\ b)$$

1.2 Options

A value of option type τ **option** is either a value v of type τ (indicated by value some v) or a distinguished value none. Option types are used in practical functional programming languages: in OCaml option types

are written 'a option, and in Haskell the option type is Maybe a. In Java, all class types are essentially option types: a value of class C may be either null or an object of class C.

We will extend the syntax of a call-by-value lambda calculus with option types by adding none and some e as new expressions. We will also provide a construct to use option values: case e_1 of $e_2 \mid e_3$, where e_1 should evaluate to an option value, e_2 and e_3 are functions, and e_2 will be applied to the unit value $()$ if e_1 evaluates to none, and e_3 will be applied to value v if e_1 evaluates to some v.

Expressions	$e ::= \cdots \mid \mathsf{none} \mid \mathsf{some}\ e \mid \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3$
Values	$v ::= \cdots \mid \mathsf{none} \mid \mathsf{some}\ v$
Types	$\tau ::= \cdots \mid \tau\ \textbf{option}$
Evaluation contexts	$E ::= \cdots \mid \mathsf{some}\ E \mid \mathsf{case}\ E\ \mathsf{of}\ e_2 \mid e_3$

We can think of type τ **option** as being syntactic sugar for the sum type **unit** $+ \tau$, and none and some e as being syntactic sugar for $\mathsf{inl}_{\textbf{unit}+\tau}\ ()$ and $\mathsf{inr}_{\textbf{unit}+\tau}\ e$ respectively.

2 Algebraic structures

2.1 Monoids

A *monoid* is a set T with a distinguished element called the *unit* (which we will denote u) and a single operation $multiply : T \to T \to T$ that satisfies the following laws.

$$\forall x \in T.\ multiply\ x\ u = x \qquad \text{Left identity}$$
$$\forall x \in T.\ multiply\ u\ x = x \qquad \text{Right identity}$$
$$\forall x, y, z \in T.\ multiply\ x\ (multiply\ y\ z) = multiply\ (multiply\ x\ y)\ z \qquad \text{Associativity}$$

The first two laws indicate that the unit u is the identity (or unity) for $multiply$. The third law says that the multiply operation is associative. The third law may look more familiar or natural if we use *infix* notation for the $multiply$ operator, instead of the prefix notation we have been using so far:

$$\forall x, y, z \in T.\ x\ multiply\ (y\ multiply\ z) = (x\ multiply\ y)\ multiply\ z$$

For those familiar with abstract algebra, a monoid is essentially a group without an inverse operator.

If we regard the set of elements T as the set of values of a specific type τ, and the operator $multiply$ as being a function of type $\tau \to \tau \to \tau$, then we have several examples of monoids readily at hand.

- Integers with multiplication. Here T is the set of values with type **Int** (i.e., the integer literals), and the $multiply$ operation is the function $\lambda a : \textbf{Int}.\ \lambda b : \textbf{Int}.\ a \times b$ (or, we can just treat $\times$ as an infix function), and unit is the integer 1.
- Integers with addition. Here T is the set of values with type **Int**, and the $multiply$ operation is the function $\lambda a : \textbf{Int}.\ \lambda b : \textbf{Int}.\ a + b$, and unit is the integer 0. Note that the same underlying set can have different operations that satisfy the monoid laws!
- Strings with concatenation. Here T is the set of values with type **String**, unit is the empty string, and the $multiply$ operation is the function $\lambda a : \textbf{Int}.\ \lambda b : \textbf{Int}.\ a{+}{+}b$, where ++ denotes string concatenation.
- Lists with append. Let τ be some fixed type, and T is the set of values with type τ **list**, i.e., lists with elements of type τ. The unit value is the empty list $[\,]$, and the $multiply$ operation is the function that appends two lists, defined above.

2.2 Functors

A *functor* is an algebraic structure that associates with each set A a set T_A. (To foreshadow how we will interpret the computational content of functors, think of T as a type constructor.)

A functor has a single operation $map : (A \to B) \to T_A \to T_B$ that takes a function from A to B and an element of T_A and returns an element of T_B. The operation map satisfies the following laws, where $\fatsemi$ denotes function composition (i.e., $(f \fatsemi g)(x) = g(f(x))$).

$$\forall f \in A \to B, g \in B \to C.\ (map\ f) \fatsemi (map\ g) = map\ (f \fatsemi g) \qquad \text{Distributivity}$$
$$map\ (\lambda a\!:\!A.\ a) = (\lambda a\!:\!T_A.\ a) \qquad \text{Identity}$$

The first law says that composing $map\ f$ and $map\ g$ (and thus getting a function from T_A to T_C) is equivalent to using map on the composition of f and g. The second law says that map of the identity function $\lambda a\!:\!A.\ a$ must be the identify function.

We can think of T_A as being a container that contains elements from the set A. Given a function that transforms elements of A to elements of B, the operator map gives us a way to transform containers of A to containers of B.

What examples do we have of functors in the programming language constructs we have seen so far?

- Options. Here T is the type constructor **option**, and a suitable function for the operation map is the following function.

  $$\lambda f\!:\!\tau_1 \to \tau_2.\ \lambda a\!:\!\tau\ \textbf{option}.\ \mathsf{case}\ a\ \mathsf{of}\ \lambda x\!:\!\textbf{unit}.\ \mathsf{none} | \lambda v\!:\!\tau.\ \mathsf{some}\ \ (f\ v)$$

  Note that if a is none, then the result of $map\ f\ a$ is none. Otherwise, if a is some v, then the result is some $(f\ v)$.

  Exercise: check to make sure that the laws are indeed satisfied.

- Lists. Here T is the type constructor **list**, and as the name suggests, a suitable function for the map operator is the map function that you may be familiar with from OCaml's standard list library.

  $$\mathsf{map} \triangleq \lambda f\!:\!\tau_1 \to \tau_2.\ \mu m\!:\!\tau_1\ \textbf{list} \to \tau_2\ \textbf{list}.\ \lambda a\!:\!\tau_1\ \textbf{list}.\ \textbf{if}\ \mathsf{isempty?}\ a\ \textbf{then}\ a\ \textbf{else}\ (f\ \mathsf{head}\ a)\ ::\ m\ (\mathsf{tail}\ a)$$

  Exercise: check to make sure that the laws are indeed satisfied.

2.3 Monads

Like functors, monads associate each set A with a set M_A (think of M as a type construtor). The computational intuition behind a monad is that M_A represents a computation that, if it terminates, will produce a value of type A, and in addition, the computation may produce some side effects. Side effects may include reading or writing memory, or throwing an exception.

A monad has two operations, called $return$ and $bind$. Operation $return$ has signature $A \to M_A$ and $bind$ has signature $M_A \to (A \to M_B) \to M_B$. For the moment, let's ignore the laws that these operators should satisfy and instead focus on the intuition behind these operators.

Operation $return$ takes a value of type A, and returns an element in M_A that represents the value. Operation $bind$ can be thought of as doing the opposite: it takes a monadic value of type M_A, and a function from A to M_B, it "extracts" the underlying value of type A from the monadic value, and passes it to the function, and produces a result of type M_B.

It's worth noting that if you flip the arguments of bind and you get something that looks very close to functor. Indeed, every monad is a functor. (Exercise: why? Given a monad with operations $return$ and $bind$, show how to define the functor operation map.)

Let's look briefly at the monad laws.

$$\forall x \in A, f \in A \to M_B.\ bind\ (return\ x)\ f = f\ x \qquad \text{Left identity}$$
$$\forall am \in M_A.\ bind\ am\ return = am \qquad \text{Right identity}$$
$$\forall am \in M_A, f \in A \to M_B, f \in B \to M_C.\ bind\ (bind\ am\ f)\ g = bind\ am\ (\lambda a\!:\!A.\ bind\ (f\ a)\ g) \qquad \text{Associativity}$$

These laws look a lot like the laws for monoid. This isn't a coincidence. For those interested in digging into such things, monads are monoids in the category of endofunctors.

Option monad. Let's consider an example of a monad: the option type. For the purposes of this examples, let's assume that the value none represents a failure, that is, the computation was unable to produce a value of type τ (think of it as a very simple kind of exception, indicating that the computation couldn't produce a value, but terminated abnormally instead). In this setting, failure of the computation is a side effect: something that may happen during the computation that is in addition (or instead of) the final value of type τ.

Let's instantiate the signatures of $return$ and $bind$ for the option monad.

$$\mathsf{return} : \tau \to \tau\ \textbf{option}$$
$$\mathsf{bind} : \tau_1\ \textbf{option} \to (\tau_1 \to \tau_2\ \textbf{option}) \to \tau_2\ \textbf{option}$$

The return operation is defined as the function that simply takes a value x of type τ and returns some x.

$$\mathsf{return} \triangleq \lambda x\!:\!\tau.\ \mathsf{some}\ x$$

The bind operation is defined as follows.

$$\mathsf{bind} \triangleq \lambda am\!:\!\tau_1\ \textbf{option}.\ \lambda f\!:\!\tau_1 \to \tau_2\ \textbf{option}.\ \mathsf{case}\ am\ \mathsf{of}\ \lambda x\!:\!\textbf{unit}.\ \mathsf{none} \mid \lambda a\!:\!\tau_1.\ f\ a$$

Let's consider what bind does. Its first argument, ma, is a value of type τ_1 **option**, that is, either a value of type τ, or none, indicating failure. Function f takes a value of type τ_1, and does some computation using that value; if the computation fails, then f will return none; otherwise, f will return a τ_2 value. Function bind combines ma and f in the obvious way: if ma is none, then the whole thing evaluates to none, otherwise, it takes the value a of type τ_1 and runs the computation represented by f.

While the option monad gives an example of what $bind$ and $return$ mean computationally, it isn't yet clear why monads are an interesting and useful computational abstraction. To further understand the benefits of monads, let's consider their use in the Haskell programming language.

3 Algebraic Structures in Haskell

Haskell is a pure functional language. Haskell has a "call by need" evaluation order, also known as *lazy evaluation*. Like call-by-name evaluation order, function arguments do not need to be evaluated to values before the function is applied, and the argument is only evaluated if it is used in the function body. Unlike call-by-name, call-by-need semantics evaluates an argument at most once (where as call-by-name may evaluate an argument expression multiple times).

3.1 Type classes

Haskell has *type classes*: a mechanism to enable ad hoc polymorphism (see Lecture 14). A type class declares common functions that all types within that class have.

We can use type classes to express the algebraic structures we defined earlier. For example, here is the declaration of the type class for monoids. The declaration says that every type g that is a member of the Monoid type class has values mempty and mappend, with types g and $\mathsf{g} \to \mathsf{g} \to \mathsf{g}$ respectively. These values correspond to the unit value and $multiply$ operation respectively. (Note that the type class does not express the laws that should hold on mempty and mappend.)

```
class Monoid g where
  mempty :: g
  mappend :: g → g → g
```

We can tell Haskell that a given type is a member of a class type by declaring it an instance of the class. The following code says that the types Int and String are monoids, and provides appropriate definitions of mempty and mappend.

```
instance Monoid Int where
 mempty = 0
 mappend x y = x + y
 -- mappend = (+)

instance Monoid String where
  mempty = ''
  mappend = (++)
```

We can write code that uses the overloading of functions provided by type classes. The following function mconcat has type [g] → g (i.e., it is a function from a list of g to a g), provided that type g is an instance of the type class Monoid.

```
mconcat :: (Monoid g) ⇒ [g] → g
mconcat [] = mempty
mconcat (x:xs) = x `mappend` mconcat xs
```

When we use mconcat, the Haskell type checker will ensure that the constraint Monoid g is satisfied, and moreover, will use the type-class instance declarations to determine the appropriate values to use for mempty and mappend. Thus Haskell is providing overloading of mempty and mappend.

```
mconcat [''hi'', ''there'']
mconcat [3,4,35]
```

Similarly, we can declare a type class that expresses the monad operations, and declare types to be in that type class.

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```

The $bind$ operator is written >>=, and used infix.

```
instance Monad Maybe where
  return = Just
  (>>=) e f =
        case e of
          Nothing → Nothing
          Just x → f x
```

3.2 Using monads

Let's see how we can use monads in computation. Let's consider a simple example where we have a list of pairs of Strings and Integers, representing a map from names to ages. Given two names (which may or may not be in the map), we want to compute the difference in ages. Let's first see how we could write this function without using monads. (We don't define the function lookup, but it has type a → [(a, b)] → Maybe b, and returns Nothing if the name isn't found in the list that represents the map.)

```
ageDiff :: String → String → [(String, Integer)]→ Maybe Integer
ageDiff n1 n2 ages =
  case lookup n1 ages of
    Nothing → Nothing
    Just a1 →
      case lookup n2 ages of
        Nothing → Nothing
        Just a2 →
          Just (abs (a1 – a2))
```

This code is quite verbose. Each time we look up the age from a name, we have to handle the case where the name wasn't present in the map. Let's use the fact that Maybe is a monad, and take advantage of the *bind* operation.

```
ageDiff' :: String → String → [(String, Integer)] → Maybe Integer
ageDiff' n1 n2 ages =
  lookup n1 ages
  >>=
    \ a1 → lookup n2 ages
  >>=
    \ a2 → return (abs (a1 – a2))
```

Here, the *bind* operation took care of chaining together the computations, and only executed the next computation if the previous one did not fail.

Haskell provides convenient syntax for using monads. We will cover this syntax a bit more in section, but it allows us to write the example as

```
ageDiff'' :: String → String → [(String, Integer)] → Maybe Integer
ageDiff'' n1 n2 ages = do {
    a1 ← lookup n1 ages;
    a2 ← lookup n2 ages;
    return (abs (a1 – a2))
  }
```

3.3 Why are monads so useful in Haskell?

Monads are central to practical programming in Haskell. There are several reasons why monads are so useful in Haskell.

1. Haskell is a pure language: programs can not have arbitrary side effects. This can be very useful. Consider an OCaml function (OCaml is not a pure language); when you execute this function, you have no idea whether the function will print something out to the screen, whether it may write to a file, whether it may throw an exception, etc. The type signature of the OCaml function does not describe what the function does, simply the value that it may return. By contrast, in Haskell, a function from, say, integers to integers, is guaranteed to have no side effects when you execute it. In Haskell, an expression with type Integer → Integer is not allowed to have any side effects.

   But side effects are useful! We may want to write a function that performs input and output, or that may raise an exception, or that may fail and return Nothing. Haskell's type system allows side effects through the use of monads. **In Haskell, monadic types cleanly and clearly express the side effects that a computation may have.**

2. **Monads force computation into sequence.** To use the *return* and *bind* operations of a monad requires choosing a sequence for computation. In the ageDiff examples above, using the monad requires us to specify that lookup n1 ages happens before lookup n2 ages.

   In general, forcing the programmer to specify a sequence for computation is great when the computations may have side effects (e.g., failure, writing to disk, etc.), as it ensures that there is a clearly de-

fined order that the side effects should happen in. In the `ageDiff` examples, the computation `lookup n2 ages` happens only if `lookup n1 ages` does not fail.

3. Type classes enable us to capture the underlying structure of the computational pattern used in monads, and Haskell provides convenient syntax to generically exploit this structure. There are many monads and language features in Haskell allow us to **capture the essence of monadic computation, and write reusable, readable code for monads.**