Class 19 - State: The State Monad

We have seen that introducing state and mutation complicates the operational semantics of our language because the memory state now becomes part of the input and output of our small-step reduction relation. Even for the non-imperative primitives in our language, we now have to thread the memory state through the individual evaluation steps. This increases the amount of “plumbing” we have to do in our interpreter implementation. In this section, we will study a class of data structures referred to as monads. Specifically, we will learn about the so called state monad. The state monad allows us to encapsulate the additional computational overhead for threading the memory state in our interpreter and to avoid exposing this complexity in the actual implementation of the step relation.

A Stateful JakartaScript Interpreter

We start our exposition with a straightforward Scala implementation of the small-step operational semantics that we formalized in class 18. As a first step, we define the abstract syntax of our language with variables and assignments using case classes, as usual. The Scala code is shown in Figure 1. We focus on the new imperative primitives of our language, the other primitives are elided. The type Mem represents memory states. It is defined as an alias to the type Map[Addr, Expr], which represents partial mappings from addresses to expressions.

We can now directly translate the small-step operational semantics into Scala code. Specifically, we implement the step relation \( \langle M, e \rangle \rightarrow \langle M', e' \rangle \) from the previous class by a function step with the following signature:

```scala
def step(m: Mem, e: Expr): (Mem, Expr)
```

That is, step takes the input memory state and an expression that is not yet a value, and returns the new memory state and the one-step reduced input expression.

In Figure 2, we show a stub of the implementation of the step function, as well as the function iterateStep, which implements the evaluation loop of the interpreter. We only show the implementation of the DoPLUS rule and
sealed abstract class Expr

/** mutabilities */
sealed abstract class Mut
case object MConst extends Mut
case object MVar extends Mut

/** declarations */
case class Decl(mut: Mut, x: String, e1: Expr, e2: Expr) extends Expr

/** binary operators */
case class BinOp(bop: Bop, e1: Expr, e2: Expr) extends Expr

sealed abstract class Bop
case object Times extends Bop
case object Plus extends Bop
...
case object Assign extends Bop

/** unary operators */
case class UnOp(uop: Uop, e1: Expr, e2: Expr) extends Expr

sealed abstract class Uop
...
case object Deref extends Uop
...

/** addresses */
case class Addr(a: Int) extends Expr

/** memory states */
type Mem = Map[Addr, Expr]

Figure 1: Representing in Scala the abstract syntax of JAKARTASCRIPT with variables and assignments
def step(m: Mem, e: Expr): (Mem, Expr) = match e with {
  /** base cases (do rules) */
  case BinOp(Plus, Num(n1), Num(n2)) =>
    (m, Num(n1 + n2))
  ...
  /** inductive cases (search rules) */
  case BinOp(bop @ (Plus|Times|...), e1, e2) =>
    val (mp, e1p) = step(m, e1)
    (mp, BinOp(bop, e1p, e2))
  ...
}

def iterateStep(e: Expr): Expr = {
  def loop(m: Mem, e: Expr): (Mem, Expr) =
    if (isValue(e)) return (m, e)
    else {
      val (mp, ep) = step(m, e)
      loop(mp, ep)
    }
  val (_, ep) = loop(Mem.empty, e)
  ep
}

Figure 2: Partial implementation of the step function and the evaluation loop of the interpreter

the SEARCHBOP₁ rule in step. The code closely follows the corresponding inference rules of the small-step reduction relation. As we can see, the additional threading of the memory state m dilutes the simplicity of the stateless implementation of our purely functional JAKARTASCRIPT variants. The goal of today’s class is to restore that simplicity.

Monads and for Expressions in Scala

As a precursor for the simplification of our stateful interpreter, we study Scala’s for expressions. The for expression primitive provides a generic way for iterating over collections of data and for building new collections from existing ones. The following example shows how a for expression can be used to iterate over a list l to build a new list by applying some function to the elements of l:

scala> val l = List(2,5)
l: List[Int] = List(2,5)

scala> for (x <- l) yield x + 1

3
res0: List[Int] = List(3, 6)

When we look at the result of the above for expression, we can see that it is really computing a map over the list l. In fact, the Scala compiler simply rewrites the for expression

```scala
for (x <- 1) yield x + 1
```
to the following expression, which calls map on l with an anonymous function that is built from the yield part of the for expression:

```scala
l map (x => x + 1)
```

Thus, a for expression is really just syntactic sugar for a call to map.

One useful feature of for expressions is that they can be nested. As an example, the following nested for expression computes the “Cartesian product” of the list l with itself:

```scala
scala> for (x <- l; y <- l) yield (x, y)
res1: List[(Int, Int)] = List((2,2), (2,5), (5,2), (5,5))
```

If we naively expand this for expression to map calls as described above, we obtain the following result:

```scala
scala> l map (x => l map (y => (x, y)))
res2: List[List[(Int, Int)]] = List(List(2,2), List(2,5), List(5,2), List(5,5))
```
The result value res2 is a list of list of pairs, rather than a list of pairs. In order to obtain res1 from res2, we need to flatten res2 by concatenating the inner lists to a single list of pairs. Incidentally, the List class provides a method flatten that does just that:

```scala
scala> res2.flatten
res3: List[(Int, Int)] = List((2,2), (2,5), (5,2), (5,5))
```

For convenience, the class List also provides a method flatMap that corresponds to the composition of map and flatten. Using flatMap we can express the nested for expression that produced res1 more compactly as follows:

```scala
scala> l flatMap (x => l map (y => (x, y)))
res4: List[Int] = List((2,2), (2,5), (5,2), (5,5))
```

This translation pattern generalizes to for expressions of arbitrary nesting depth. In general, the Scala compiler will translate a for expression of the form

```scala
for (xn <- en; ...; x2 <- e2; x1 <- e1) yield e0
```
to the expression

```scala
en flatMap (xn => ... e2 flatMap (x2 => e1 map (x1 => e0)))...
```
Monads. The use of for expressions is not restricted to the List type. It works for any type that provides a map and a flatMap method with the appropriate signatures. For example, the Option type also provides these functions and can hence be used in for expressions:

```scala
scala> for (x <- Some(0)) yield x + 1
res5: Option[Int] = Some(1)

scala> for (x <- None) yield x + 1
res6: Option[Int] = None
```

We refer to a class that has appropriate map and flatMap methods as a monad. One can think of a monad as an abstract data type that implements a container for data and provides generic functions for transforming this data without extracting it from the container. Using for expressions we can then conveniently express a sequence of such transformations that operate directly on the contained data.

The monad-as-container correspondence is easy to see for the type List and also for the type Option. The latter can be thought of as a list of length at most 1. In general, monads can be more abstract and it is sometimes more difficult to understand the nature of the contained data. We will see one such example in the next section. Many of the classes that are provided by the Scala standard API are monads, including all of the collection classes. Some of the more interesting monads provided by Scala are Try and Future.

The Theory of Monads. As an aside, the term monad is lent from category theory, a branch of mathematics that is concerned with the theory of mathematical structures and the morphisms between them. The programming language and category theoretic concepts of a monad are closely related. In category theory, monads are defined in terms of certain algebraic laws that relate the flatMap and map functions. For example, these laws codify how map can be expressed in terms of flatMap and vice versa. These laws also ensure that for expressions in Scala behave the way they are expected to behave.

The State Monad

To simplify the implementation of our stateful interpreter given in Figure 2, we introduce the state monad. The state monad allows us to hide the threading of the memory state between the different calls to the step function. In other words, the state monad takes care of the plumbing so that we can focus our attention on the interesting bits of the interpreter implementation. Before we introduce the state monad, we first refactor the interpreter given in Figure 2 into a semantically equivalent version that will help us understand what it is that the state monad actually does.

Curried Interpreter. The transformation that we apply to the interpreter is as follows: we modify the signature of the step function from:
def step(e: Expr): Mem => (Mem, Expr) = match e with {
  /**< base cases (do rules) */
  case BinOp(Plus, Num(n1), Num(n2)) =>
    { m => (m, Num(n1 + n2)) }
...
  /**< inductive cases (search rules) */
  case BinOp(bop @ (Plus|Times|...), e1, e2) =>
    { m =>
      val (mp, ep) = step(e1)(m)
      (mp, BinOp(bop, ep, e2))
    }
...
}

def iterateStep(e: Expr): Expr = {
  def loop(e: Expr): Mem => (Mem, Expr) =
    if (isValue(e)) return { m => (m, e) }
    else {
      { m =>
        val (mp, ep) = step(e)(m)
        loop(ep)(mp)
      }
    }
  val (_, ep) = loop(e)(Mem.empty)
  ep
}

Figure 3: Curried version of the interpreter shown in Figure 2
def step(m: Mem, e: Expr): (Mem, Expr)
to
    def step(e: Expr): Mem => (Mem, Expr)

More precisely, we turn step into a curried function that first takes the input expression e and then returns a new function. This new function can then be applied to the input memory state m to compute the output state and one-step reduced expression (mp, ep), as before. From a caller’s point of view, the step function behaves exactly as before, except that every call step(m, e) has to be replaced by step(e)(m). The new curried implementation of the interpreter is shown in Figure 3. The new version looks more complicated than before. Though, as is often in life, things first have to get worse before they get better.

Implementing the State Monad. The state monad is a parameterized type State[S, R]. The essence of State[S, R] is that it is a container that encapsulates a function of type S => (S, R), which can be seen as a computation that takes an input state of type S and returns an output state together with a result value of type R. Looking at the result type Mem => (Mem, Expr) of the curried step function, we can see that it is precisely a computation of this form. That is, we can encapsulate the result of a call step(e) in a state monad State[Mem, Expr]. The map and flatMap functions of this monad will then take care of threading the memory state m through a sequence of calls to step.

The implementation of the state monad is shown in Figure 4. The encapsulated computation of type S => (S, R) is held in the field run, which is a parameter of the class State. We discuss the map and flatMap methods in more detail.

The map method transforms the given State[S, R] into a State[S, P] by composing a stateless computation f: R => P with the encapsulated stateful computation run. It is instructive to compare the body of the map method with the implementation of the SEARCHBINOP1 rule in step shown in Figure 3. We can see that the latter can be obtained from the former by substituting run by step(e) and f by \{ e1p => BinOp(bop, e1p, e2) \}. That is, the implementation of SEARCHBINOP1 corresponds to a call to map on the result of step(e1) encapsulated in a state monad.

Similar to map, the flatMap method transforms the given State[S, R] into a State[S, P] by composing run with another computation f. The difference to map is that now f is itself stateful and returns a state monad. If we again carefully analyze the code in Figure 3, we observe that the else branch of the body of loop in iterateStep corresponds to a call to flatMap on the result of step(e) encapsulated in a State[Mem, Expr]. Here, step(e) takes the role of run in flatMap and loop takes the role of f.

Finally, we add a companion object for the State class that provides a factory method apply for inserting a result value r into a state monad. The encapsulated computation simply returns r together with the unmodified input state s.
sealed class State[S,R](run: S => (S,R)) {
  def apply(s: S) = run(s)

  def map[P](f: R => P): State[S,P] =
    new State[S,P]({ s =>
        val (sp, r) = run(s)
        (sp, f(r))
    })

  def flatMap[P](f: R => State[S,P]): State[S,P] =
    new State[S,P]({ s =>
        val (sp, r) = run(s)
        f(r)(s) // same as f(r).apply(s)
    })
}

object State {
  def apply[S,R](r: R): State[S,R] =
    new State[S,R]({ s => (s,r) })
}

Figure 4: Implementation of the state monad

def step(e: Expr): State[Mem, Expr] = match e with {
  /** base cases (do rules) */
  case BinOp(Plus, Num(n1), Num(n2)) =>
    State(Num(n1 + n2))
  ...
  /** inductive cases (search rules) */
  case BinOp(bop @ (Plus|Times|...), e1, e2) =>
    step(e1) map { e1p => BinOp(bop, e1p, e2) }
  ...
}

def iterateStep(e: Expr): Expr = {
  def loop(e: Expr): State[Mem, Expr] =
    if (isValue(e)) return State(e)
    else step(e) flatMap loop

  val (_, ep) = loop(e)(Mem.empty)
  ep
}

Figure 5: Monadic version of the interpreter shown in Figure 3
def step(e: Expr): State[Mem, Expr] = match e with {
  /** base cases (do rules) */
  case BinOp(Plus, Num(n1), Num(n2)) => State(Num(n1 + n2))
  ...
  /** inductive cases (search rules) */
  case BinOp(bop @ (Plus|Times|...), e1, e2) =>
    for (e1p <- step(e1)) yield BinOp(bop, e1p, e2)
  ...
}

def iterateStep(e: Expr): Expr = {
  def loop(e: Expr): State[Mem, Expr] =
    if (isValue(e)) return State(e)
    else for (ep <- step(e); epp <- loop(ep)) yield epp
  val (_, ep) = loop(e)(Mem.empty)
  ep
}

Figure 6: Variant of the interpreter shown in Figure 5 with for expressions

Using the State Monad. With the implementation of the state monad in place, we can refactor the interpreter in Figure 3 as shown in Figure 5. In particular, we are using map and flatMap to thread the memory state through the recursive calls to step as described above. The plumbing is now completely hidden inside of the state monad. Note that the rule DoPlus in step is implemented with a call to the apply method of State’s companion object. To simplify our implementation further, we can additionally hide the map and flatMap calls using for expressions as shown in Figure 6.