Latest: HOWTO: Get tenure
Next: Canvas tag rendering of mathematical functions
Prev: First-class macros from meta-circular evaluators
Rand: Productivity tips for academics

Architectures for interpreters: Substitutional, denotational, big-step and small-step

[article index] [email me] [@mattmight] [+mattmight] [rss]

Software architectures for interpreters are rarely covered in college courses. Given that many programmers will find themselves implementing a scripting language or a domain-specific language for their company some day, I think this absence partly explains the profusion of poorly implemented internal scripting languages. The closest one gets to a class on interpreters is usually in a programming language semantics class. It's accurate enough to say that a semantics is just an interpreter that's not allowed to use side effects. (Or, you could think of an interpreter as a semantics that's allowed to use side effects.) There are four popular architectures for semantics---substitutional, denotational, big-step and small-step. When taken as architectures for interpreters, each has pros and cons in terms of performance, flexibility and implementation complexity. This article compares these architectures through the lens of the same minimalist, higher-order dynamic language: the lambda calculus. (Code examples are provided in highly-commented Scala.)

Resources

- Structure and Interpretation of Computer Programs. Until very recently, MIT used to teach freshman computer science by having them write interpreters. SICP is the now-classic textbook for that course.
- Lisp in Small Pieces is a complete, up-to-date treatment of writing high-performance interpreters and compilers for dynamic languages (with Scheme used as an example).
- My post on meta-circular evaluators and first-class macros includes a complete meta-circular interpreter for a large chunk of Scheme.

- My recommended reading for programming languages.

A simple language: Lambda calculus

The lambda calculus is one of the simplest Turing-complete programming languages, containing only three expression types: variable references, function calls, and anonymous functions. Its BNF grammar has just three rules:

```
<exp> ::= <variable>                    [references]
        |  (<exp> <exp>)                [function calls]
        |  (lambda (<variable>) <exp>)  [anonymous functions]
```

Please pardon the Lispish syntax; I have a habit of using simple syntax when syntax isn't the focus. Rest assured that all of this still applies to languages like Python, JavaScript and Ruby. We can define abstract syntax node types as Scala classes and then use parser combinators to parse programs in this language:

[Exp.scala]

```
  def unapply(e : Exp) : Option[Exp] = e match {
    case Ref(_) | Lambda(_,_) => Some(e)
    case _ => None
  }
}



/**
 A parser for a toy lambda-calculus-like language.
 */
object ExpParser extends JavaTokenParsers {

  def expr : Parser[Exp] = ref | lambda | app

  def lambda : Parser[Exp] = "(" ~ "lambda" ~ "
(" ~ ident ~ ")" ~ expr ~ ")" ^^
        { case "(" ~ "lambda" ~ "
(" ~ id ~ ")" ~ e ~ ")" => Lambda(id,e) }

  def app : Parser[Exp] = "(" ~ expr ~ expr ~ ")" ^^
        { case "(" ~ f ~ a ~ ")" => App(f,a) }

  def ref : Parser[Exp] = ident ^^
        { case id => Ref(id) }

  def parse(reader : java.io.Reader) : Exp = parseAll(expr,reader).get
```

Substitution-based interpreters

Substitution-based interpreters model an executing program's state as a program, and they make progress by repeatedly transforming one program into another. Substitutional interpreters based on textual substitution or raw string manipulating (e.g. Tcl) are notoriously inefficient. *Tree*-based

substitutional interpreters can achieve reasonable efficiency, but their real strength lies in their straightforward implementation.

Substitutional interpreters can become constricting when used to implement more complex language features. Recursion, dynamic allocation and side effects, in particular, can be onerous. These interpreters also require special care to avoid exponential blow-up in code size during the reduction process.

A tree-based substitutional interpreter re-arranges the abstract syntax tree of the program during interpretation. A tree-based substitutional interpreter for the lambda calculus uses one observation to make progress:

`((lambda (`*v*`) ... `*v*` ...)` *argument*`) == ...` *argument* `...`

The formalized version of this observation---that when you apply a function, you can replace occurrences of its parameter with the argument---is called β-reduction.

In Scala, the interpreter boils down to a mechanism for performing "binding-aware" substitutions on expression trees, and a mechanism for performing β-reduction. A binding-aware substitution makes sure that it won't accidentally change the meaning of an expression when it performs a substitution. For example, if we try to substitute the expression `(f x)` in place of the variable `v` in the body of the function `(lambda (x) ...)`, then the value of `x` will be "captured" by the formal parameter inside the lambda term, when it's supposed to belong to the outer binding of the variable `x`.

[SubstitutionInterpreter.scala]

```
  def interpret (exp : Exp) : Exp = {
    val exp_ = reduce(exp)
    if (exp != exp_)
      interpret(exp_)
    else
      exp
  }

  /**
   Traces an execution.
   */
  def trace (exp : Exp) : Exp = {
    println(exp)
    val exp_ = reduce(exp)
    if (exp != exp_)
      trace(exp_)
    else
      exp
  }


  // This runs to a fixed-point:
  def test1 =
    trace(ExpParser.parse("((lambda (f) (f f)) (lambda (g) (g g)))"))

  // This would run forever in another interpreter, but stops here
  // when we hit the same term twice:
```

Denotational interpreters

The philosophy behind denotational interpretation is to encode the interpreted language's features as the equivalent features in the host language. (Meta-circular interpreters take this approach to the extreme by defining a language in terms of itself.) In the study of semantics, the host language is mathematics.

If the purpose of the interpreter is to allow users of some program to write plug-ins, the tight coupling between the interpreted language and the host language makes denotational interpreters attractive: if done right, the program won't be able to tell the difference between compiled procedures and interpreted procedures.

On the other hand, if the interpreted language contains features not present in the host language, then the cleanliness and simplicity of this approach starts to break down.

A denotational interpreter for the lambda calculus only has to map each expression to a value, and in the pure lambda calculus, the only kind of value available is the function. Of course, an expression, by itself, may not perfectly define a function; for example, the meaning of

```
(lambda (x) (f x))
```

depends on what the value of the variable `f` is. Because this term contains a free variable, it is called an **open** term. To turn this expression into a value, it must be coupled with a dictionary that determines the value of its free variables; that dictionary is called an **environment**. When an open term is paired with an environment that defines all of its free variables, that pair is called a **closure**. A closure unambiguously determines a procedure.

In an implementation, the fundamental run-time data structure in a denotational interpreter of the lambda calculus is the closure:

[DenotationalInterpreter.scala]

```
/**
 In the pure lambda calculus, the only kind of value is a function.
 */
case class Fun (f : D => D) extends D {
  def apply(d : D) : D = f(d)
}

/**
 Environments map variables to values.
 */
type Env = Map[Variable,D]

/**
 Transforms an environment and an expression into a value.
 */
def eval (env : Env) (e : Exp) : D = e match {
  case Ref(v)      => env(v)
  case App(f,e)    => (eval(env)(f)) (eval(env)(e))
  case Lambda(v,e) => Fun(d => eval(env(v) = d)(e))
```

```
  }

  /**
   Converts an expression into a value.
   */
  def interpret (exp : Exp) : D = eval (Map()) (exp)
}
```

Big-step interpreters

A big-step interpreter is a kind of operational interpreter. An operational interpreter is one that focuses on moving from one machine state to the next. (A substitutional interpreter is actually a special kind of operational interpreter in which the state of interpretation is encoded as a program itself.) The core of an operational interpreter is a transition function that takes in a machine state and returns the next machine state(s). In a big-step interpreter, the transition function isn't required to terminate on all inputs, and it is generally recursive.

A big-step interpreter for the lambda calculus uses expression closures for machine states, and it "reduces" one expression closure into another. For example, if the current machine state is a closure containing a variable v and an environment $env : Variable \rightarrow Value$, then the reduction of this state is the closure $env(v)$.

Big-step interpreters don't tie themselves to implementing features of the interpreted language as features in the host language. As a result, they don't chafe when trying to implement features not present in the host language. On the other hand, they don't make interaction between the host language and the interpreted language quite as natural.

In Scala, a big-step interpreter for the lambda calculus has roughly the same implementation complexity as the denotational interpreter:

[BigStepInterpreter.scala]

```
    def reduced : ExpClo = exp match {

      // Closures over lambda terms are final.
      case Lambda(_,_) => this

      case Ref(v) => env(v) // Already reduced in call-by-value.

      case App(f,e) => {
        val fClo = ExpClo(f,env).reduced
        val eClo = ExpClo(e,env).reduced
        val ans = fClo match {
          case ExpClo(Lambda(v,body),env2) => ExpClo(body,env2(v) = eClo)
          case _ => throw new Exception("error: applying non-
procedure: " + fClo)
        }

        ans.reduced
      }
    }
```

```
}


/**
 Reduces an expression to its final state.
 */
```

Small-step interpreters

A small-step interpreter is an operational interpreter which factors computation into guaranteeably computable chunks. The core of a small-step interpreter is a transition function that is guaranteed (by the implementer) to terminate for any input. Interpreting a program consists of repeatedly applying the transition function. Small-step interpreters have the highest implementation complexity, but they make it possible to implement complex language features (like threads or continuations) or to perform debugging by tracing. Because small-step interpreters don't leverage the host language's stack, it is possible to write a garbage collector for a small-step interpreter. (Denotational and big-step interpreters have to trust the host language's garbage collection; substitutional interpreters can get away with reference-counting on trees.)

The keys to implementing a small-step interpreter for the lambda calculus are **value-expression contexts** and **continuations**. A value-expression is a hybrid syntactic/semantic expression-like value, in which some of its sub-trees are values instead of expressions. A continuation, in this case, is just a model of the program's run-time stack.

Value-expressions and their contexts

Suppose an interpreter wants to evaluate an expression like `(+ a (* b c))`. It will first evaluate the sub-expression `a` into a value. The interpreter will then look up `a` in the current environment to find its value is (perhaps) *3*. Then, the current expression becomes a value-expression, with a semantic value (noted with italics) inside it: `(+ 3 (* b c))`. Next, it will evaluate `b` and then `c`, by which point the current value-expression will look something like: `(+ 3 (* 4 5))`. At this point, it has enough information to evaluate `(* 4 5)`, turning the top-level value-expression into `(+ 3 20)`. Then, the whole program evaluates into the value *23*. All of the intervening states of computation between `(+ a (* b c))` and *23* are value-expressions.

A value-expression context is a value-expression with a "hole" in it, representing a place to drop in some value. For example, if evaluating the value-expression `(+ 3 (* b c))`, once the interpreter decides to evaluate `b` next, it can remember where it was by creating a context: `(+ 3 (* □ c))`. Then, it can evaluate the sub-expression `b` into the value *4* and drop the result into the "hole" to create a new value-expression context.

Continuations

When a language involves procedures, evaluating a value-expression might switch evalution to some other value-expression inside another procedure. So, now a simple value-expression context by itself isn't rich enough to remember where the computation was before evaluating a sub-expression. Continuations are like a context for an entire computation, and they are equivalent to a run-time stack. A continuation is a list of value-expression contexts paired with environments. Once the current value-expression is fully resolved to a value, the head of that list is popped off, and the value is dropped into the hole before resuming execution.

Implementing contexts

There's a data structure for modeling contexts precisely called a zipper, but a small-step interpreter doesn't need an actual context structure. For any value-expression object, we can define two functions: *next-sub-expression* and *insert-value*. The *next-sub-expression* function returns the next unevaluated sub-expression, and the *insert-value* function returns a new value-expression with the "next" sub-expression to be evaluated replaced by a value.

[SmallStepInterpreter.scala]

```
    Thrown when trying to insert into an uninsertable value-expressions.
    */
  case class BadInsertionException extends Exception

  /**
   Thrown when trying to get the next expression to evaluate in a value-
expression where all sub-terms are evaluated.
   */
  case class NoNextExpressionsException extends Exception

  /**
   Thrown once the machine halts.
   */
  case class Halted(finalValue : D) extends Throwable

  /**
   Continues to take steps until no more steps can be taken.
   */
  def interpret (exp : Exp) : D = {
    try {
      var state = State(exp,Map(),HaltCont)
      while (true) {
        state = next(state)
      }
      throw new Error("There is a bug in the Matrix.")
    } catch {
      case Halted(d) => d
```

Prev: First-class macros from meta-circular evaluators
Rand: Productivity tips for academics