Integrating Formal Models into the Programming Languages Course

Allen B. Tucker
Computer Science Department
Bowdoin College
Brunswick, ME 04011
allen@bowdoin.edu

Robert E. Noonan
Computer Science Department
College of William and Mary
Williamsburg, VA 23187
noonan@cs.wm.edu

Abstract

In our approach to the Programming Languages course, formal models are integrated as a thread that pervades the course, rather than as a one of many topics that is introduced and then dropped. In this paper we present a simplified version of the formal semantics used. We then show how this model is implemented using the object-oriented, functional and logic paradigms.

1 Introduction

The study of programming languages at the advanced undergraduate or graduate level usually covers two main areas: principles of language design and several different programming paradigms. Texts for this course take one of two approaches: 1) concept-based surveys of a wide range of language design topics and paradigms; and 2) interpreter-based treatments of the design principles presented in a functional language.

Our approach to the course attempts to unite the best features of these two approaches into a single and coherent framework. Like the interpreter-based texts, we include a rigorous, complete, and hands-on treatment of the principles using a formal grammar, type system, and denotational semantics, including an interpreter that implements the formal model. Like the concepts-based texts, our approach presents and contrasts the programming paradigms; however, in doing so, we have recurring examples that appear in each major paradigm, one of which is an interpreter that implements the formal model. Our approach is more fully described in our own text [6].

Our approach is based on the belief that a formal treatment of syntax and semantics, a consistent use of the mathematical notations learned in discrete mathematics, and a hands-on treatment of the principles of language design are centrally important to the study of programming languages.

This approach is advocated, for instance, in the design of the Programming Languages course in the "Liberal Arts Model Curriculum" [7], and is consistent with the recommendations of *Computing Curricula 2001* [2]. We believe that the formal treatment of semantics should not be presented as one of many unrelated topics in a way that encourages students to dismiss it altogether. We think that a better approach is one which integrates the formalism in a more compelling way. This strategy is consistent with recent calls for better integration of mathematical rigor into the core computer science curriculum [1, 5].

Our treatment of syntax and semantics includes the use of BNF grammars and a formal denotational approach to type systems and run-time semantics. This approach is fully integrated, so that the theory can be explored by students with the aid of interpreters that directly implement the formal semantics. Such an approach allows students to study all the dimensions of language design using the available formal tools: BNF grammars, abstract syntax, recursive descent parsing, and functional definitions of type systems and meaning. We use a small imperative language named "Jay" as a basis for illustrating the principles of language design and formal methods.

In this paper we illustrate our approach by first giving a formal semantic model for integer expressions in Jay. In the sections which follow, we implement this model using the object-oriented, functional and logic paradigms. This approach has been used successfully at both of our schools despite differences in the courses that precede the programming languages course.

2 Formal Semantics of Expressions

The definition of a programming language is complete only when its syntax, type system, and run-time semantics are fully defined. The *semantics* of a programming language is a definition of the *meaning* of any program that is syntactically valid from both the concrete syntax and the static type checking points of view. One way to formally define the semantics of a programming language is to define the meaning of each type of statement that occurs in the (abstract) syntax as a state-transforming mathematical function. Thus, the meaning of a program can be expressed as a collection of functions operating on the program state. This approach is called *denotational semantics*.

Three useful semantic domains for programming languages are the "environment," the "memory," and the "locations." The *environment* γ is a set of pairs that unite specific variables with memory locations. The *memory* μ is a set of pairs that unite specific locations with values. The *locations* in each case are the natural numbers **N**.

The *state* σ of a program is the product of its environment and its memory. However, for our purposes in this paper, it is convenient to represent the state of a program in a simpler form that takes the memory locations out of play and simply defines the *state* σ of a program as a set of pairs $\langle v, val\rangle$ that represents a binding of an active variable *v* and its currently-assigned value *val*.

Let Σ represent the set of all program states σ. Then a meaning function M is a mapping from a particular member of a given element (or class) of the abstract syntax and current state in Σ to a new state in Σ:[1]

$$M\colon\ Class \times \Sigma \to \Sigma$$

For instance, the meaning of an abstract *Statement* can be expressed as a state-transforming function of the form.

$$M\colon\ Statement \times \Sigma \to \Sigma$$

These functions are necessarily *partial* functions, which means that they are not well-defined for *all* members of their domain $Class \times \Sigma$. That is, the abstract representations of certain program constructs in certain states do *not* have finite meaning representations, even though those constructs are syntactically valid.

In this paper, we only consider the meaning of a Jay *Expression* in abstract syntax, so that we can make some fairly simple assumptions about the types of its arguments and result. We assume that an *Expression* has only `int` arguments and only the arithmetic operators + and *. Thus, all individual arithmetic operations and results are of type `int`. This discussion uses the following notational conventions:

- $v \in \sigma$ tests whether there is a pair whose *Identifier* is *v* in the current state σ
- $\sigma(v)$ is a function that extracts from σ the value in the pair whose *Identifier* is *v.*

The abstract syntax of a language is a formal device for identifying the essential syntactic elements in a program without describing how they are concretely constructed. Consider, for example, the Pascal and C/C++ while loop statements. Clearly, at an abstract level they are the same, consisting of a loop test and a loop body. Syntactic differences are nonessential to the fundamental looping process that they represent.

The abstract syntax of a programming language can be defined using a set of rules of the following form:

$$Lhs = Rhs$$

where *Lhs* is the name of an *abstract* syntactic *class* and *Rhs* is a list of essential components that define a member of that class. Each such component has the form of an ordinary declaration, identifying the member and its name. The individual components are separated by semicolons (;). For example, consider the abstract syntax of a *Loop* like the one described above.

$$Loop = Expression\ \text{test};\ Statement\ \text{body}$$

This definition defines the abstract class *Loop* as having two components, a `test` which is a member of the abstract class *Expression*, and a `body` which is a member of the abstract class *Statement.*

For the purposes of this paper, we will focus on the meaning of expressions whose abstract syntax is:

$$\begin{aligned} Expression &= Variable \mid Value \mid Binary \\ Binary &= Operator\ \text{op};\ Expression\ \text{term1, term2} \end{aligned}$$

In this abstract syntax for expressions, a *Variable* is a string representing an identifier and a *Value* is an integer.

Using the abstract syntax for *Expression* given above, we can define the meaning of an *Expression* as follows:

$$\begin{aligned} &M:\ Expression \times State \to Value \\ &M(Expression\ \text{e}, State\ \sigma) \\ &\quad = e \qquad \text{if e is a Value} \\ &\quad = \sigma(e) \qquad \text{if e is a Variable} \\ &\quad = Apply(\text{e.op}, M(\text{e.term1}, \sigma), M(\text{e.term2}, \sigma)) \qquad \text{if e is a binary} \end{aligned}$$

The Jay type system requires that all variables used in a program be declared; therefore, there cannot be a reference to a variable that is not in the state. This is checked by the type system, which is not discussed here.

Next, we define the auxiliary function *Apply*, which takes two integer values and calculates an integer result. Here is its definition for the arithmetic operators:

$$\begin{aligned} &Apply:\ \text{Operator} \times Value \times Value \to Value \\ &Apply(Operator\ op, Value\ v1, Value\ v2) \\ &\qquad = v1 + v2 \qquad \text{if op is a +} \\ &\qquad = v1 \times v2 \qquad \text{if op is a *} \end{aligned}$$

Implementing the denotational semantics of a programming language provides a ready test bed for experimentation with semantic models using languages in different programming

1. Some treatments of formal semantics define the meaning of a program as a series of functions in which the name (e.g., $M_{Statement}$) distinguishes it from the rest, e.g., $M_{Statement}\colon \Sigma \to \Sigma$.

language paradigms. In our courses we cover a more extensive model, including both integer and boolean values, all 4 arithmetic operators, the logical operators, and the boolean operators, as well as various forms of statements including assignment, block, if's, while's, and the skip statement.

Having the students explore an implementation of the formal model in each of Java, Scheme, and Prolog is very instructive. First, it serves to help them better understand the various aspects of the semantic model. Second, having the formal model implemented in each paradigm helps the students to better understand the strengths and weaknesses of each paradigm. In the sections which follow we provide implementations of the model presented above in each of the languages Java, Scheme, and Prolog.

3 Object-Oriented Implementation

A fundamental decision in implementing the formal model is the representation of a state. Since a state σ is a set of unique key-value pairs, the state of a computation is naturally implemented as a subclass of a Java `Hashtable.` The functional expression $\sigma(v)$, which extracts from σ the value of variable *v*, can be implemented in Java using the `Hashtable` method `get` as:

```
sigma.get(v)
```

One approach to implementing the semantics of expressions and statements requires defining a Java class `Semantics` which contains a method for each of the various functions *M* and auxiliary functions (like *Apply*) that together define the meaning of a program. The methods in this class refer to objects that are defined by the abstract syntax. The abstract syntax thus serves as a bridge between the concrete syntax of a program and its meaning.

The meaning *M* of an arithmetic *Expression* with operators + and * follows directly from its definition as well. Since no state transformations occur here, all we are asking is that the Java method return a *Value* (rather than a *State*). The following Java implementation is suggested:

```
Value M (Expression e, State sigma) {
    if (e instanceof Value)
       return (Value)e;
    if (e instanceof Variable)
         return (Value)(sigma.get((Variable)e));
    if (e instanceof Binary)
       return apply(((Binary)e).op,
           M(((Binary)e).term1, sigma),
           M(((Binary)e).term2, sigma));
    return null;
}
```

Implementation of these meaning functions *M* in Java is relatively painless and straightforward.

An alternative approach is to use a more object-oriented style to build a set of classes to mimic the structure of the abstract syntax. One way to achieve this aim is to make *Expression* the parent or superclass, with each of the elements of the abstract syntax a subclass. In this way, we can discuss the concept of *ad hoc polymorphism* in contrast to true polymorphism in which we ask an *Expression* for its meaning without regard to whether the *Expression* is a *Variable*, a *Value*, or a *Binary*. Such a class hierarchy is depicted below.

```
abstract class Expression {
    abstract public Value M(Expression e,
        State sigma);
}
class Value extends Expression { ... }
class Variable extends Expression { ... }
class Binary extends Expression { ... }
```

Now the question arises: what methods should each of these concrete classes have? Clearly each class needs a constructor and an *evaluate* or meaning method *M*. In presenting these classes, we shall omit, for the sake of brevity, any other methods.

As before, the meaning of a *Variable* is simply its value in the current state. Such a meaning function is easily added to the class *Variable*, as show below. Because this is being done in an object-oriented manner, the expression *e* is the receiver of the message, unlike the mathematical expression where *e* is an explicit argument to the meaning function.

```
class Variable extends Expression {
    private String id;
    public Variable(String id) { this.id = id; }
    public Value M(State sigma) {
        return (Value)(sigma.get(this));
    }
}
```

Note that *hashcode* and *equals* methods must also be added to class *Variable*. Implementing the *Value* class involves a little more analysis, but is similar.

In the abstract syntax *Binary* serves as a grouping mechanism, in that every binary expression is either an instance of an add, a multiply, etc. One possible implementation is to make the *Binary* class abstract, with specific subclasses for *AddBinary*, *MultiplyBinary*, etc. An alternative is to make the operator into an abstract class, with specific subclasses for *AddOp*, *MultiplyOp*, etc.; in this alternative, the mathematical *apply* function would be a method in which each operator implements its specific semantics or meaning.

In keeping with the underlying mathematics, we choose the latter approach. In this case, the implementation of the meaning function is straightforward; first each term is asked for its meaning and then the operator is asked to apply itself to the resulting values.

```
class Binary extends Expression {
  Operator op;
  Expression term1, term2;
  public Binary(Operator o, Expression t1,
                 Expression t2) {
    op = o;  term1 = t1;  term2 = t2;
  }
  public Value M(State sigma) {
    return op.apply(term1.M(sigma),
                    term2.M(sigma));
  }
}
```

The Operator class and the related AddOp, MultiplyOp, etc. classes are equally straightforward. The only method needed in class Operator is apply, which must be implemented by the concrete subclasses.

```
abstract class Operator {
  public abstract Value apply(Value v1, Value v2);
}
class AddOp extends Operator {
  public Value apply(Value v1, Value v2) {
    return new Value(v1.intVal() + v2.intVal());
  }
}
class MulitplyOp extends Operator {
  public Value apply(Value v1, Value v2) {
    return new Value(v1.intVal() * v2.intVal());
  }
}
```

The world of the *Expression* class and its subclasses are an illustration of the *Command* design pattern [4, 3], in which each of the subclasses of *Expression* is responsible for evaluating itself and returning the appropriate response. The Command pattern is heavily used in graphics applications, in which different graphics objects are asked to draw or paint themselves on the screen; the invoking method does not know or care how the painting is done.

4 Functional Implementation

In this section we implement the formal semantics of Jay expressions using Scheme. A state in Scheme is naturally represented as a list, with each element of the list being a pair representing the binding of a variable to its value. So the Jay state: $\{\langle x, 1\rangle, \langle y, 5\rangle\}$ can be represented as the Scheme list:

```
((x  1) (y  5))
```

Next we implement the state access functions named `get` from the Java implementation (see Section 3). Recall that the `get` function is used to obtain the value of a variable from the current state.

```
(define (get id sigma)
   (if (equal? id (caar sigma))) (cadar sigma)
       (get  id  (cdr sigma))
))
```

Since the Jay type system requires that all variables used in a program be declared, there cannot be a reference to a variable that is not in the state and there is no need to check for a null list.

Next, we consider the Scheme meaning function for Jay expression evaluation. To this end, we choose an appropriate tagged list representation for an abstract Jay expression:

```
;;; (value number)
;;; (variable  ident)
;;; (operator term1  term2)
;;;    where operator is one of: plus times
```

The meaning function for a Jay abstract expression is implemented as a case on the kind of expression. The meaning of a value expression is just the value itself. The meaning of a variable is the value associated with the variable in the current state. The meaning of a binary expression is obtained by applying the operator to the meaning of the operands:

```
(define (m-expression expr sigma)
  (case (car expr)
        ((value) (cadr expr))
        ((variable) (get (cadr expr) sigma)
        (else (apply (car expr) (cadr expr)
                 (caddr expr) sigma))
))
```

The function `apply` is implemented as a case on the operator. Again we show only the two arithmetic operators:

```
(define (apply op term1 term2 sigma)
  (case op
      ((plus)  (+ (m-expression term1 sigma)
                  (m-expression term2 sigma)))
```

```
     ((times) (* (m-expression term1 sigma)
                    (m-expression term2 sigma)))
     (else #f)
))
```

5 Logic Paradigm Implementation

In this section we implement the formal semantics of Jay expressions using Prolog. A state here is naturally represented as a list, with each element of the list being a pair representing the binding of a variable to its value. So the Jay state $\{\langle x, 1\rangle, \langle y, 5\rangle\}$ can be represented as the Prolog list:

```
[[x,1], [y,5]]
```

Next we have to implement the state access function named `get`, which in Java was used to obtain the value of a variable from the current state. The `get` function takes an input variable and an input state and produces a output value.

```
/* get(var, inState, outValue) */
```

The base case is that the variable-value pair occur at the front of the state list, in which case the value associated with the variable is the desired result value. Otherwise the search continues through the tail of the list; astute students note the similarity to the Scheme implementation.

```
get(Var, [[Var, Val] | _], Val).
get(Var, [_ | Rest], Val) :- get(Var, Rest, Val).
```

Next, we consider the function for the meaning of a Jay expression. To this end, we choose an appropriate representation for a Jay expression in abstract syntax. One possibility is to use lists; instead we prefer using structures:

```
/*  value(number)
    variable(ident)
    operator(term1, term2),
        where operator is one of: plus times */
```

The meaning of a Jay abstract expression is implemented as a set of rules depending on the kind of expression. In Prolog, these rules take an input expression and an input state and return a value:

```
/* mexpression(expr, state, val) */
```

The meaning of a `value` expression is just the value itself.

```
mexpression(value(Val), _, Val).
```

The meaning of a `variable` is the value associated with the variable in the current state, obtained by applying the `get` function.

```
mexpression(variable(Var), State, Val) :-
   get(Var, State, Val).
```

The meaning of a binary expression is obtained by applying the operator to the meaning of the operands; below we show the meaning for `plus`:

```
mexpression(plus(Expr1, Expr2), State, Val) :-
   mexpression(Expr1, State, Val1),
   mexpression(Expr2, State, Val2),
   Val is Val1 + Val2.
```

This definition says first evaluate `Expr1` in `State` giving `Val1`, then evaluate `Expr2` in `State` giving `Val2`. Then add the two values giving the resulting value. The remaining binary operators are implemented similarly.

6 Conclusion

In our approach to the Programming Languages course, formal semantics is a thread that is introduced early. It uses the basic mathematical ideas of functions, logic and proof that students learned in the Discrete Mathematics (Discrete Structures) course, thus reinforcing the integral nature of mathematics in computer science.

This thread is continued through each of the major programming language paradigms, in which an implementation of the formal model is presented in class and used as a source of homework exercises. We believe that such an approach is more principled and more effective for students, compared with the alternative of presenting an isolated unit on formal semantics and then ignoring it throughout the rest of the course.

References

[1] Baldwin, Doug. How mathematical thinking enhances CS problem solving. *SIGCSE Technical Symposium on Computer Science Education*, (March 2001), pp. 390-391.

[2] *Computing Curricula 2001: Computer Science* (August 2001), www.computer.org

[3] Cooper, James W. *Java Design Patterns*. Addison-Wesley, 2000.

[4] Gamma, Erich, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley.

[5] Tucker, Allen et al. Our curriculum has become math-phobic! *SIGCSE Technical Symposium on Computer Science Education*, (March 2001), pp. 243-247.

[6] Tucker, Allen and Robert Noonan. *Programming Languages: Principles and Paradigms*, McGraw-Hill, 2002.

[7] Walker, Henry M. and G. Michael Schneider. Revised model curriculum for a liberal arts degree in Computer Science. *Communications of the ACM*, 39 (December 1996), pp. 85-95.