

Automatic Generation of Compilers and Interpreters from Object-Oriented Specifications of Programming Languages

Barrett R. Bryant and Viswanathan Vaidyanathan

Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Blvd.
Birmingham, Alabama 35294-1170, U. S. A.
{bryant, vaidyana}@cis.uab.edu

Abstract. An object-oriented formal specification methodology for programming languages is proposed which allows for automatic generation of compilers and interpreters in an object-oriented manner. Our specification approach constructs an object-oriented representation of denotational semantics and it is this representation which not only facilitates the specification process, but also allows for implementation of the specifications. Object-oriented specification allows for the abstracting of various common details among programming languages so that formal syntax and semantics can be defined elegantly. This allows for specifications of one language to be developed and then reused to produce specifications of related languages, both at the level of syntax and semantics. Our implementation strategy is to transform the specification into code expressed in the Java programming language and this code may be produced as either an interpreter or a compiler. Since these implementations are automatically developed from the formal specifications, a mechanism is achieved for compiler/interpreter reuse at the specification level.

1 Introduction

The formal specification of programming languages and automatic construction of language implementations from these specifications have been studied for a number of years. Advantages of such automatic implementation are: 1) these implementations may serve as prototypes for evaluating the language specification during design, 2) compiler front-ends can now be completely automated in what was previously a code-intensive and sometimes error-prone task, 3) compilers built using specification-based approaches are more likely to be “correct”

Proceedings of the Twenty Second Australasian Computer Science Conference, Auckland, New Zealand, January 18–21 1999. Copyright Springer-Verlag, Singapore. Permission to copy this work for personal or classroom use is granted without fee provided that: copies are not made or distributed for profit or personal advantage; and this copyright notice, the title of the publication, and its date appear. Any other use or copying of this document requires specific prior permission from Springer-Verlag.

than those built using informal descriptions of the language, and 4) front-ends produced by such approaches are portable across many different platforms and have a standardized target language that may be interpreted or compiled into the language of the underlying machine. Despite these attractions, most compilers are still constructed by hand after initial automation of the lexical and syntactic analysis phases. This is due to the fact that formal specifications of the later phases of the compiler tend to use notations which are often difficult to use or understand, and the back-end parts of compilers which are produced automatically tend to be substantially less efficient than those coded by hand. In this paper, we address these deficiencies by providing a specification methodology suitable for automatic construction of all phases of compilers. Our approach is to define an object-oriented representation of programming language syntax and semantics which has advantages over existing techniques in its readability and understandability for the compiler writers, yet also facilitates automation in an efficient manner. Furthermore, this specification approach allows for reuse of language, and hence compiler, components at the specification level.

This paper is organized as follows. In Section 2, we briefly review formal specification of programming languages and in section 3, our object-oriented representation of programming language specification is introduced. Section 4 explains the implementation methodology which allows automatic generation of compilers and interpreters. Finally we conclude in Section 5.

2 Formal Specification of Programming Languages

In order to develop a compiler or interpreter for any programming language, we must know all the details of the language. Therefore, the formal specification of that programming language must completely describe both syntax and semantics.

The syntax of a programming language deals with the form of various expressions in the language and is formally defined using a Backus-Naur Form (BNF), or context-free, grammar. This type of notation is said to comprise the language's *concrete syntax*. In order to avoid some semantically irrelevant features such as operator precedence and associativity present in concrete syntax, an *abstract syntax* is usually used as the basis for formal semantics specifications. An abstract syntax specifies just the compositional structure of a program omitting some aspects of their concrete representation as strings of symbols, thus providing a much simpler syntax. It is straightforward to define *syntax-directed translations* from concrete syntax into abstract syntax. Furthermore, these translations can be automated by automatic parser generation tools (e.g. see [Appel, 1998]).

Techniques to represent the semantics of programming languages are not as well established as those available for specifying the syntax of languages. The three most commonly used methods to formally describe the meanings of programs are operational semantics, denotational semantics and axiomatic semantics. We chose denotational semantics [Schmidt, 1986] because of its unique

combination of mathematical properties and formalization of execution models. The denotational semantics of a language maps a program in that language directly to its meaning, represented as a mathematical value called its *denotation*. The primary components of a denotational semantics specification are the *semantic algebras* and *valuation functions*. The semantic algebras define a collection of semantic domains and the valuation functions map syntax domains into functions over semantic domains, usually expressed in λ -calculus [Hindley and Seldin, 1986]. This mapping is an abstract compiler since the denotational code produced is executable, e.g. using ML [Milner et al., 1990] to interpret it or through compilation into real machine code. There are several well-known techniques for converting the specification of this mapping into executable compilers, usually termed *automatic compiler generation* [Lee, 1989].

3 Object-Oriented Specification of Semantics

Even though denotational semantics techniques are mathematically sound, and can be used in semantics-directed compiling, they suffer from lack of modularization, reusability and readability. We have developed an object-oriented specification of programming languages that supports reuse of specification in defining the semantics of various languages, with the ultimate goal being semantics-directed compiler generation. Such compilers will be constructed in an object-oriented manner, allowing for not only reuse of compiler source code in developing new compilers, but even reuse at the specification level. There are techniques for constructing compilers in an object-oriented manner [Holmes, 1995] and tools for constructing compilers in object-oriented programming languages such as Java [Appel, 1998]. Since specifications of languages may be converted into executable compilers and interpreters, a framework for reuse of such specifications should be a more elegant approach than source-code compiler reuse as in [Ancona et al., 1994] and [Weber, 1992]. For the attribute grammar formalism, this has been demonstrated by the Eli compiler generation system [Kastens and Waite, 1994].

Our object-oriented specification mechanism abstracts out the various common features of programming language syntax and semantics to an appropriate level so that the semantic translation from syntactic phrases of any language can be given in an elegant fashion. We specify the language by describing syntax and semantic domains with their associated operations. Many features of language specification in the valuation functions are abstracted out into the semantic algebra or made inherent properties of the syntax domains. We develop the syntax and semantic domains and associated operations in an object-oriented fashion so that it is easy to inherit and specialize them depending upon the semantic necessities of the language. Such a framework aids in the reusability of the semantic features of different languages to build new languages and new language implementations.

Our general approach is as follows:

- Every syntax and semantic domain is treated as a class.
- Operations on domains are limited to the ones explicitly defined (i.e., all domain components are private to the class and its subclasses).
- Domains may be constructed by aggregation or as subclasses of other domains by inheriting. We are also developing other constructs which will be useful in extending the semantics in an object-oriented fashion.

The general concept of our idea was introduced in [Bryant and Vaidyanathan, 1998]. Here we give more precise details leading up to the automatic generation of compilers and interpreters.

3.1 Syntax Domains

Syntax domains are defined by the abstract syntax rules of the specification to represent the various syntactic components of the language (e.g. declarations, expressions, statements, etc.). Each of these aspects may be represented as a proper domain in an object-oriented style. To consider just a specific case, the “statement” domain will represent the features that are common for all statements. Then specific statements will inherit all the properties of statements. For example, imperative language statements inherit the functionality of mapping states to states. Structures like *for*, *while*, *do-while*, etc., can inherit and modify from a general “loop” domain. Examples of syntax domain specifications are:

```
Statement S ::= A | L
Assignment A ::= V = E
Loop L ::= while C do S | do S while C | for A1; C; A2 do S
```

For brevity, we omit specifications of Variables (V), Expressions (E) and Conditions (C). These grammar notations can be interpreted in an object-oriented manner by regarding the Statement as a “class” which consists of objects in classes Assignment and Loop, i.e. Assignment *is-a* Statement and Loop *is-a* Statement and hence both inherit properties of statements. Assignment is an aggregate of a Variable and Expression. Likewise, there are three different classes of Loop.

In organizing our specification in an object-oriented manner, we have elected to consider syntax domains as the primary classes of the specification. Since the valuation functions map the syntax domains into functions over semantic domains, we consider these as operations over the syntax domains. Valuation functions can then be described elegantly by calling the appropriate operations of the syntax domains. We illustrate this idea with an example. Below is the standard functional denotational semantics of assignment:

$$S \llbracket \mathbf{V} = \mathbf{E} \rrbracket = \lambda \text{state} . \text{update} (\mathbf{V} \llbracket \mathbf{V} \rrbracket) (\mathbf{E} \llbracket \mathbf{E} \rrbracket \text{state}) \text{state}$$

This semantics indicates that we evaluate expression \mathbf{E} using the state and then update the variable \mathbf{V} in the existing state to return a new state. For simplicity in this presentation, we assume that \mathbf{E} has no side effects and so returns only a simple value.

The object-oriented version of this specification is:

$$\llbracket \mathbf{V} = \mathbf{E} \rrbracket . S = \lambda \text{state} . (\text{state} . \text{update} (\llbracket \mathbf{V} \rrbracket . V) (\llbracket \mathbf{E} \rrbracket . E (\text{state})))$$

Note that the main components of the specification are still present, but the orientation has changed from a function being applied to arguments into a property of a syntax domain object. This specification may be used to represent assignment statements in almost any language because of inheritance and polymorphism. For example, we may define a class of “states” with an update method. In its simplest form, this state may be a mapping of variables to values (a combination of environment and store). Through inheritance this may be extended to represent 1) a mapping of variables to locations (the environment), coupled with a mapping of locations to values (the store), 2) an environment, and a stack of stores, 3) an environment, stack of stores, and collection of files for input and output, etc. Throughout all of these cases, the basic semantic definition of assignment remains the same. We may also inherit the semantics of this assignment to handle assignments to subscripted variables as well as assignments of actual to formal parameters in procedure and function calls. Finally, we may specialize the specification for languages where the assignment returns a value or the expression has a side effect.

Another example of an object-oriented semantic definition is in the specification of looping. We first define the most general loop statement:

$$\begin{aligned} \llbracket \mathbf{L} \rrbracket . S &= \text{loop} \\ &\text{where loop} = \text{fix } (\lambda f . \lambda C . \lambda S . \lambda \text{state} . \\ &\quad \text{if } \llbracket \mathbf{C} \rrbracket . C (\text{state}) \text{ then } f (\llbracket \mathbf{S} \rrbracket . S (\text{state})) \text{ else } \text{state}) \end{aligned}$$

This specification indicates that the denotation of a loop is the fix-point of a function which checks the condition \mathbf{C} in the context of the current state - if the condition is true, then the function is recursively called with the state produced by evaluating the statement (body of the loop) using the current state (as with expressions, we assume that conditions have no side effects); if the condition is false, the current state is returned unchanged, thereby ending the recursive calling which effects the loop.

We can use this specification to create other concise loop semantics.

$$\begin{aligned} \llbracket \mathbf{while} \ \mathbf{C} \ \mathbf{do} \ \mathbf{S} \rrbracket . S &= \text{loop } \llbracket \mathbf{C} \rrbracket \llbracket \mathbf{S} \rrbracket \\ \llbracket \mathbf{do} \ \mathbf{S} \ \mathbf{while} \ \mathbf{C} \rrbracket . S &= \lambda \text{state} . \text{loop } \llbracket \mathbf{C} \rrbracket \llbracket \mathbf{S} \rrbracket (\llbracket \mathbf{S} \rrbracket . S (\text{state})) \\ \llbracket \mathbf{for} \ \mathbf{A}_1; \ \mathbf{C}; \ \mathbf{A}_2 \ \mathbf{do} \ \mathbf{S} \rrbracket . S &= \lambda \text{state} . \text{loop } \llbracket \mathbf{C} \rrbracket \llbracket \mathbf{S}; \ \mathbf{A}_2 \rrbracket (S (\llbracket \mathbf{A}_1 \rrbracket . S (\text{state}))) \end{aligned}$$

3.2 Semantic Algebra

Semantic algebra describes the various semantic domains and the operations associated with the elements of those domains. They are defined as compositions of mappings of other domains, including basic domains like integers, boolean, etc., and auxiliary domains like environment (which specifies the symbol table of a compiler), stores, expressible values (those returned by expressions in a program), denotable values (those denoting variables in a program), storable values (values that are stored in locations), etc. Based on their functionality, different domains will be specified so that details are abstracted out properly, and the domains lend themselves to extensions.

Semantic domains may be formally constructed using product (\times), sum ($+$) and function mapping (\rightarrow) operations. These are illustrated as follows. Note that A and B are assumed to be existing semantic domains (i.e. classes) and C the new domain we are constructing.

1. Product domains $C = A \times B$

A product domain is an aggregation of other domains. From an object-oriented viewpoint, we may define C as a class whose instance variables are from domains A and B. The principal operation of this domain is a selector for each component. The constructor is a function over the component domains, mapping to the larger domain.

2. Sum domains $C = A + B$

The sum domain differs from the product domain in that the elements of C are either elements of A or B, and are tagged to indicate which domain they belong to. Elements a and b of A and B, respectively, may be injected into or projected out of the sum domain. Since objects are already tagged with the class they belong to, the sum domain may be represented in a manner similar to the product domain except that the domain constructor will accept an argument of any component type. Selectors merely project out the specified component.

3. Function domains $C = A \rightarrow B$

A function domain is a mapping from elements of one domain to elements of another. This is usually represented by λ -abstraction. For example, $\lambda x.x + 1$ denotes a function with argument x which returns $x + 1$, i.e., the successor function over domain $\text{Integer} \rightarrow \text{Integer}$. In an object-oriented semantics, the simplest function domain will be a mapping of objects in the class to values, i. e., selector functions. Other function domains may map objects into objects of the same class, i. e., modifier functions. Finally, there may be complex function domains which combine both of these aspects. As in standard denotational semantics, we define such functions through lambda abstraction.

The above domains are all constructed through aggregation of other domains. An important special case of the product domain occurs when domains are inherited. We introduce an inherits operator \leftarrow for this purpose. If $B \leftarrow A : C$, then B has the domain structure of A with any additional components specified by C. For example, if $\text{Store} = \text{Location} \rightarrow \text{Integer}$, denoting a memory store mapping locations to integer values, then we may generalize this to a stacked store by adding a Location “stack pointer,” as in $\text{StackedStore} \leftarrow \text{Store} : \text{Location}$. Operations on the newly created domain have access to the inherited components and functions in the usual manner.

It is important to note that the object-oriented denotational metalanguage semantics remains the same as the standard denotational metalanguage, since the class hierarchies may be “flattened” into standard denotational semantics, using an approach similar to that suggested in [Palsberg and Schwartzbach, 1994].

4 Automatic Generation of Interpreters and Compilers

The denotational semantics of a programming language may be either interpreted or compiled. We may also desire mixed compilation and interpretation. In either case, we must first compile the program into the abstract syntax representation. Since we use Java as our implementation language, our approach is to use the compiler development tools designed for Java [Appel, 1998] to develop this front end, namely JLex for generating a lexical analyzer and CUP for generating a syntax analyzer with abstract syntax tree construction according to the syntax domains. Once the abstract syntax representation has been constructed, the denotational semantics may be used to “interpret” the tree using the semantic functions. This may be viewed as a two-part process, the first merely translating the syntax into a semantic representation, including with static semantics, and the second to evaluate the static semantics to produce a pure dynamic semantics representation. The latter may then be interpreted directly or compiled further into machine code. One of the main difficulties in producing efficient compilers automatically is the ability to separate static and dynamic semantics constructions. We have identified classes for static and dynamic concerns in our specification to simplify this process, according to the principles set forth in [Lee, 1989]. For example, static semantics components include the environment with associated naming and scope information about identifiers which is not needed at run-time, while dynamic semantics components include the state and addressing information about identifiers. In the type structure of a language, different components get updated at different times (e.g., declaration time or execution time), based on whether the language is typed or untyped, if there is static or dynamic typing, etc. Our semantics classifies these different issues so that the semantics-directed implementation techniques may perform static operations at compile time and dynamic operations at run time.

An overall view of this process is shown in Figure 1. Both the interpretation and compilation approaches are explored in the next two sections.

4.1 Interpretation

A denotational semantics for a language L may be used to produce an interpreter N by generating code which reads the source program P and some input I to that program and generates the output O of P on I . In this case, the denotational semantics should be thought of as representing a function $N : (P \times I) \rightarrow O$. We produce N in Java.

Considering the example syntax domains presented in the previous section, it is straightforward to see how these may be interpreted. Figure 2 shows sample Java code which may be automatically generated from the object-oriented specifications. Note that the `WhileLoop` class inherits everything from `Loop` while the `DoWhileLoop` and `ForLoop` classes modify the valuation function as given in the formal specification. We have not included any static semantic evaluation.

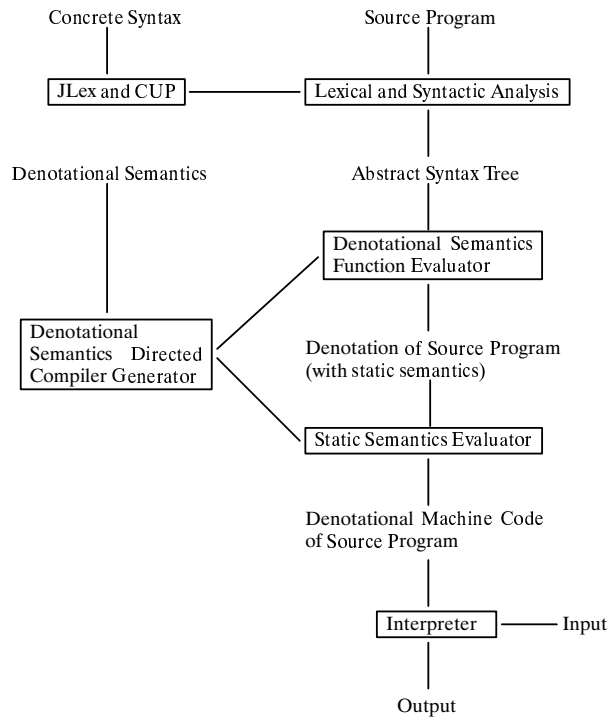


Fig. 1. Overview of Denotational Semantics-Directed Implementation

4.2 Compilation

A denotational semantics for a language L may be used to produce a compiler C by generating a representation of the code for the source program P . This “denotational machine code” will then read some input I and generate the output O of P on it I . In this case, the denotational semantics should be thought of as representing a function $C : P \rightarrow (I \rightarrow O)$. We produce C in Java and represent the denotational machine code using an object-oriented representation.

In contrast with the interpretive approach of the previous section, here the denotational semantics must be compiled into denotational machine code. Figure 3 shows the Java code for the same example presented previously which shows this difference. The code which is generated takes the form of *abstract combinator trees*. Combinators are typically denotational expressions which have direct representations in conventional target machine languages. In defining the semantics of various languages, the operations on the semantic domains can be modeled

```

abstract class Statement {
    ...
    public abstract State S (State state);
}

class Loop extends Statement {
    protected Condition c;
    protected Statement s;
    ...
    public State S (State state) {
        if c . C (state) then return S (s . S (state)); else return state;
    }
}

class WhileLoop extends Loop { ... }

class DoWhileLoop extends Loop {
    ...
    public State S (State state) { super . S (s . S (state)); }
}

class ForLoop extends Loop {
    protected Assignment as1, as2;
    ...
    public ForLoop (Assignment as1, Condition c, Assignment as2, Statement s) {
        super (c, s . compose (as2)); // add final assignment to body
        ...
    }
    ...
    public State S (State state) { super . S (as1 . S (state)); }
}

```

Fig. 2. Java Code for Interpretation

as combinators which can be modified and reused for different languages. To illustrate how this code generator works, consider the following source language statement:

```
for (i = n; i; i--) fac = fac * i;
```

The code generated for this statement from the given Java code is shown in Figure 4. It can be seen that the denotational machine code is the set of combinator expressions defined by the valuation functions of the denotational semantics itself. These combinator expressions may be further transformed into pseudo-machine code as shown in Figure 5. Specific machine variants of this code may be executed very efficiently.

```

abstract class Statement {
    ...
    public abstract Code S ();
}

class Loop extends Statement {
    protected Condition c;
    protected Statement s;
    ...
    public Code S () { return Code . fix (Code . if (c . C (), s . S ()); }
}

class WhileLoop extends Loop { ... }

class DoWhileLoop extends Loop {
    ...
    public Code S () { return Code . compose (s . S (), super . S ()); }
}

class ForLoop extends Loop {
    protected Assignment as1, as2;
    ...
    public ForLoop (Assignment as1, Condition c, Assignment as2, Statement s) {
        super (c, s . compose (as2)); // add final assignment to body
        ...
    }
    ...
    public Code S () { return Code . compose (as1 . S (), super . S ()); }
}

```

Fig. 3. Java Code for Compilation

```

compose (
    update (location [ i ]) (access (location [ n ]))
    fix (
        if (access (location [ i ]) != 0 then
            compose (
                update (location [ fac ]) (access (location [ fac ])) * (access (location [ i ]))
                decrement (location [ i ])
            )
        else
            break
    )
)

```

Fig. 4. Denotational Machine Code for for Loop

```

    n = access (location [ n ] )
    update (location [ i ] ) n
L1:
    i = access (location [ i ] )
    if i = 0 goto L2
    fac = access (location [ fac ] )
    i = access (location [ i ] )
    fac = fac * i
    update (location [ fac ] ) fac
    i = access (location [ i ] )
    i--
    update (location [ i ] ) i
    goto L1
L2:

```

Fig. 5. Machine Code for `for` Loop

5 Summary and Conclusions

Our goal was to build a suitable framework which would facilitate the reuse of specifications of the semantics of programming languages to improve the process of compiler/interpreter design. Toward this end, we have built an object-oriented specification framework for the semantics of programming languages which allows specification reuse in developing semantics-directed compilers and interpreters for a variety of programming languages. Our framework abstracts out different common details of various aspects of programming languages to construct a library of reusable components. We can inherit the different relevant features from this framework and specialize in order to derive the semantic specifications of any particular language. In our preliminary studies we first reused the semantics of Pascal to develop a C semantics, and then reused Smalltalk to develop a C++ semantics, finally reusing this C++ semantics to produce the semantics of Java. We have also successfully reused various versions of C semantic specifications to ultimately produce the semantics of C++ by “inheriting” the former and extending with object-oriented features. For example, we can first define the semantics of classes and objects to create an object-based C, and then define the semantics of inheritance to create C++. Using our approach, we were able to define the semantics of various programming language constructions and then extend and combine the functionality of these semantics to produce specifications of more powerful languages. Compilers for all of these languages may be automatically generated using the proposed approach.

Our current system has concentrated on front-end translation of source programs into “denotational machine code.” From this code, conventional optimizations and target code generation approaches may be used to generate actual machine code. We are currently incorporating these into a complete compiler generation system from the object-oriented specifications. In future work we will investigate the use of “object-oriented” ML dialects (e.g. [Reppy and Riecke,

1996]) for representing combinators. Our object-oriented framework should also facilitate reusing the combinators and we would expect to define a set of “object combinators” that would represent the various operations to be performed dynamically. It would also be interesting to explore how an object-oriented semantics specification affects *partial evaluation* [Jones et al., 1993], one method commonly used to implement semantics-directed compilers.

References

- Ancona, M., Dodero, G., and Clematis, A. (1994). Reusing a compiler. *Proc. 1994 ACM Symp. Applied Computing*, pages 82–87.
- Appel, A. W. (1998). *Modern Compiler Implementation in Java*. Cambridge Univ. Press.
- Bryant, B. R. and Vaidyanathan, V. (1998). Object-oriented specification in programming language design and implementation. *Proc. COMPSAC '98, 22nd Ann. Int. Computer Software and Applications Conf.*, pages 387–392.
- Hindley, J. and Seldin, R. (1986). *Introduction to Combinators and λ -Calculus*. Cambridge Univ. Press.
- Holmes, J. (1995). *Object-Oriented Compiler Construction*. Prentice Hall.
- Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice Hall.
- Kastens, U. and Waite, W. M. (1994). Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627.
- Lee, P. (1989). *Realistic Compiler Generation*. MIT Press.
- Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML*. MIT Press.
- Palsberg, J. and Schwartzbach, M. I. (1994). *Object-Oriented Type Systems*. John Wiley & Sons.
- Reppy, J. H. and Riecke, J. G. (1996). Simple objects for standard ml. *Proc. 1996 ACM Conf. Programming Language Design and Implementation*, pages 171–180.
- Schmidt, D. A. (1986). *Denotational Semantics - A Methodology for Language Development*. Allyn and Bacon, Inc.
- Weber, C. (1992). Creation of a family of compilers and runtime environments by combining reusable components. *Proc. 4th Int. Conf. Compiler Construction*, pages 110–124.