

# Object-Oriented Software Specification in Programming Language Design and Implementation

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 workbench is proposed for defining the syntax and semantics of programming languages, using which the formal properties of different languages can be elaborated and analyzed. Our specification approach is an object-oriented representation structured around a denotational semantics methodology, which abstracts out various common details so that formal syntax and semantics can be defined elegantly. Specific details can be inherited and specialized in defining the semantics of various programming languages, and the reusability and modifiability of many programming languages features can be manifested across language paradigms. Furthermore, prototype implementations for these languages may be automatically developed from the formal specifications, providing a mechanism for compiler/interpreter reuse at the specification level.*

## 1. Introduction

Developing the formal specification of the requirements of a software system facilitates the building of a correct and consistent software implementation of that system. Some of the important characteristics of the specifications are that they should be readable, they should be easily implementable, and they should be easily extendible. In order to develop a related system, it is highly desirable to reuse and modify as many of the specifications of the earlier system as possible. We are interested in exploring the aspect of reusability of software specifications as applied to defining the formal semantics of programming languages since there has been much success in applying formal specification methodology to programming languages syntax and semantics. Even the development of specification languages

for general software systems started with the idea of specifying programming languages [3].

In order to develop a compiler for any programming language, we need to know about the syntax and the semantics of the language. The compilers built using a semantics-based approach are more likely to be “correct” than those built using informal descriptions of the language. The different techniques to represent the semantics of programming languages are not as well established as those available for specifying the syntax of languages, but one of the most widely used methods is denotational semantics [22]. 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. Moreover when we observe the semantics of different languages, even those based on different paradigms, there are various commonalities between them which are apparent when we get beyond the syntax level. There is a need for a framework which addresses this issue and facilitates the reuse of such features.

In this paper we develop an object-oriented framework of the syntax and semantics of programming languages that serves as a specification workbench which can be effectively reused and specialized in specifying various languages. There are various domains of the syntax and semantics of languages and operations related to those domains, that can be abstracted out using an object-oriented framework. The various internal details of the domains can be properly encapsulated using this approach. Syntax and semantics of different languages can be defined using this workbench by inheriting and specializing different relevant parts. New languages can be built by combining the features of different languages. By modularizing the semantic representation, our framework also enhances its readability. This framework has been used in a design tool in the specification of languages, an educational tool in understanding the

features of different languages, and a model for how reuse may be done at the specification level in other problem domains.

This paper is organized as follows. In Section 2, we present the foundations of formal specification of programming languages and in section 3, the theory of software reuse as applied to this application is reviewed. Section 4 discussed our object-oriented representation of programming language specification, and section 5 explains the implementation methodology. Finally we conclude in Section 6.

## 2. Formal Specification of Programming Languages

The formal specification of a programming language must provide a detailed description of both its syntax and its 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 [2]).

Semantics refers to the meaning of the syntactic structures of a programming language. Traditionally semantics of a programming language are defined informally using descriptions in English. Semantics defined in this way are usually incomplete and ambiguous. Formal semantics, on the other hand, can be used to provide a precise, machine-independent, unambiguous specification of the meaning of a program. Also, formal semantics techniques provide us with rigorous theory using which we can prove properties of programs and programming languages.

For formalization of semantics, we chose denotational semantics 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. These functions are usually expressed in  $\lambda$ -calculus [9]. This mapping may be thought of as an abstract compiler since the denotational code produced is executable, e.g. using ML [17] or some other functional programming language 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* [15].

*High-level semantics* [15] was a first step towards modularizing semantics. It separated the denotational semantics specification of a language into static and dynamic parts that facilitates semantics-directed compiler generation. This approach was also achieved by *two-level semantics* [19]. The concept of *action semantics* [18] is another evolutionary step in language specification engineering since it also has a denotational semantics basis and contributes towards the modularity and readability of semantics. Cartwright and Felleisen [4] discuss a format for extending the denotational semantics of a language, represented in direct semantics, that accommodates orthogonal extensions of the language. *Evolving algebra* [7] is an improvement on the operational semantic approach to provide reusable semantic specification. Kastens and Waite [12] deal with a framework of object-oriented attribute grammar for reusable language semantics. Liang, Hudak, and Jones [16] and Steele [24] have functional approaches for improving the modularity in defining the semantics of languages, using *monads*.

To facilitate the specification of software in general, various *specification languages* are being built; VDM [3], Z [23], B [14], and LARCH [8], to name a few. Some of these languages are object-oriented and there are object-oriented extensions to the others (e.g., Object-Z [6]) in order to better facilitate abstraction and encapsulation. Of these specification language efforts, Dong, Duke, and Rose [5] use Object-Z to define the semantics of a small procedural language.

Our goal is to develop 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. Semantics-directed compiling has the advantage that the compilation being performed is consistent with the language specification, and the compiler is produced automatically, instead of by programmers.

## 3. Software Reuse and Programming Language Implementation

Software reuse is the process of creating software systems from existing software rather than building them from scratch. In order to facilitate reusability, various reusable artifacts have to be abstracted out. Different relevant components have to be selected, specialized and integrated in

order to build a new system [13]. One approach to software reuse is to build various libraries related to different applications which can be reused. Another approach is to follow an object-oriented framework in building a system, so that various features can be inherited, modified and reused effectively. Both these approaches to software reuse are done at the source-code or design level. We are interested in exploring reusability at the specification level.

There are two ways in which reuse can be effected in the development of a software system.

1. Horizontal reuse refers to the process of software reuse where various components of one system are reused in its transformation to another related system.
2. Vertical reuse refers to the reuse done in the refinement or enhancement of one system. This can be done in a rapid prototyping environment of product enhancement.

Compilers can be constructed in an object-oriented manner to facilitate reuse [10]. When a compiler is developed for any language, there are many different components that can be reused from other compilers. Some work has been done in reusing components of a compiler for one language to develop a compiler for another language [1], but this is reuse at the source-code level. Weber [25] discusses a system which modularizes various components in a language environment by decoupling them, so that they can be put together and linked to build a system, but that also does not deal with specification reuse. Since there are well-known techniques for even automatically converting specifications of languages to executable compilers and interpreters, a framework for reuse of such specifications should be a more elegant approach than source-code compiler reuse.

## 4. Object-Oriented Specification of Semantics

The denotational semantics approach does not lend itself easily to extendibility. There are too many details at the semantic function level where the syntactic constructs are mapped to corresponding denotations. The internal details of the structure of various semantic domains are exposed in the semantic definition. There is need for modularizing various features of programming languages while specifying the semantics.

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 can be abstracted out into the semantic algebra, in order to provide a more elegant semantic function definition. Furthermore, these can be made inherent properties of the syntax domains. We can 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.

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.

### 4.1. Syntax Domains

Syntax domains are defined by the abstract syntax rules of the specification to represent the various syntactic components of the language. This includes declarations, expressions, statements, etc. We represent each of these aspects as proper domains which can be laid out in an object-oriented fashion. 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, in an imperative language, this means that statements will inherit the functionality of mapping states to states. Another example of such inheritance occurs with looping structures in a language like *for*, *while*, *do-while*, etc., which can inherit and modify from a general “loop” domain.

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. If we lay out the syntax domains and their operations properly then valuation functions can be described elegantly by calling the appropriate operations of the syntax domains. As an example, a specification of assignment statements may be used in almost any language through 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.

## 4.2. 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 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  $\text{Location}$  “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 [20].

## 5. Semantics-Directed Implementation of Programming Languages

The denotational semantics of a programming language may be either interpreted or compiled. The former is preferred for more dynamic languages while the latter is preferred for more static languages. We may also desire mixed compilation and interpretation. In either case, we must first

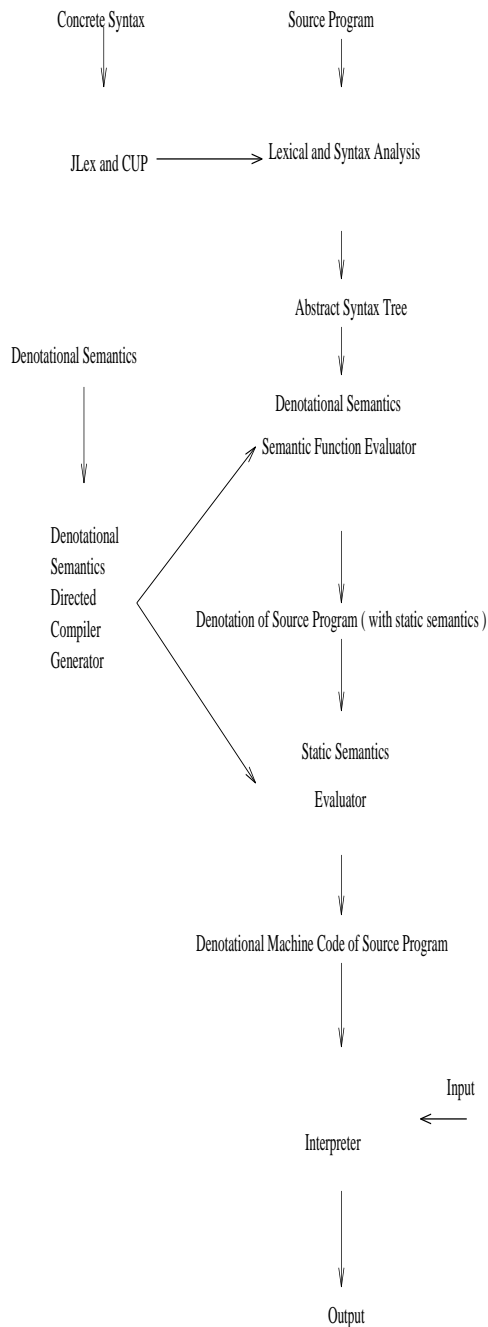
compile the program into the abstract syntax representation. Since we use Java as our implementation language, our approach is to use the compiler development tools designed for Java [2] to develop this front end, namely JLex for generating a lexical analyzer and CUP for generating a syntax analyzer 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.

In the interpretive approach, a denotational semantics for a language  $L$  is 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 a Java. That is, an interpreter written in Java is automatically generated from the object-oriented specification.

In the compilation approach, a denotational semantics for a language  $L$  is 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. 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. That is, a compiler written in Java is automatically generated from the object-oriented specification.

In contrast with the interpretive approach, here the denotational semantics must be compiled into denotational machine code. 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 as combinators which can be modified and reused for different languages. Specific machine variants of this code may be executed very efficiently.

An overall view of this process is shown in Figure 1.



**Figure 1. Overview of Denotational Semantics-Directed Implementation**

## 6. Summary and Conclusions

An object-oriented framework for programming language design is proposed which supports reusability of software specification, especially in the field of *semantics-directed compiler generation*. What we are doing can be considered as a compiler reuse, at a specification level as opposed to source-code level. We believe that this approach will have a positive impact in the field of semantics of programming languages by being a good design tool. As languages develop richer and richer semantics, e.g. Java, it can be very beneficial to have such a tool that facilitates the formalization of semantics. Java is an excellent candidate for such study since its semantics are not currently formally defined. Our method can also contribute as an educational tool in understanding the various aspects of programming languages. We have used the semantic notations presented here to explain formal semantics to undergraduate students and found that the object-oriented approach to semantics provided a much clearer conceptual understanding of language semantics than traditional denotational semantics approaches. It was particularly straightforward for the students to move from a formal specification to a programmed implementation using this approach. Our approach should also serve as a model for how reuse may be done at the specification level in other problem domains.

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 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. Horizontal reuse has been achieved in our preliminary studies by reusing the semantics of Pascal to develop a C semantics, and reusing Smalltalk to develop a C++ semantics, and then reusing this C++ semantics to produce the semantics of Java. Vertical reuse has been achieved by reusing 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++. We have also done some experiments with the semantics of concurrency in this regard. Using our approach, we are 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.

Several issues which were addressed in this research are:

- Differences in scoping structures of languages, including the scopes of variables, functions, etc., impact the effectiveness of inheritance.
- 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.
- Language features may be specified using a *direct* or *continuation* semantics approach.

Areas for future work include the following:

- We will investigate the use of “object-oriented” ML dialects (e.g. [21]) 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.
- *Partial evaluation* [11] is one method commonly used to implement semantics-directed compilers. It will be interesting to explore how an object-oriented semantics specification affects partial evaluation.
- Our approach should also facilitate enhancing a particular language by inheriting other features. A useful example would be to inherit features of object-oriented and concurrent languages to get the semantics of a concurrent object-oriented language.

## References

- [1] M. Ancona, G. Dodero, and A. Clematis. Reusing a compiler. *Proc. 1994 ACM Symp. Applied Computing*, pages 82–87, 1994.
- [2] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge Univ. Press, 1998.
- [3] D. Björner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice Hall, 1982.
- [4] R. Cartwright and M. Felleisen. Extensible denotational semantics. *Proc. Symp. Theoretical Aspects of Computer Software*, pages 244–272, 1994.
- [5] J. S. Dong, R. Duke, and G. Rose. An object-oriented approach to the semantics of programming languages. *Proc. 17th Australian Computer Science Conf.*, pages 767–775, 1994.
- [6] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. *Proc. TOOLS 5 - Technology of Object-Oriented Languages and Systems*, pages 465–483, 1991.
- [7] Y. Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.
- [8] J. V. Guttag and J. J. Horning. *LARCH: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
- [9] J. Hindley and R. Seldin. *Introduction to Combinators and  $\lambda$ -Calculus*. Cambridge Univ. Press, 1986.
- [10] J. Holmes. *Object-Oriented Compiler Construction*. Prentice Hall, 1995.
- [11] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [12] U. Kastens and M. W. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
- [13] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.
- [14] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer Verlag, 1996.
- [15] P. Lee. *Realistic Compiler Generation*. MIT Press, 1989.
- [16] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. *Proc. 22nd ACM Symp. Principles of Programming Languages*, 1995.
- [17] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [18] P. D. Mosses. *A Tutorial on Action Semantics*. Notes for Formal Methods Europe, 1994.
- [19] H. R. Nielson and F. Nielson. Semantics directed compiling for functional languages. *Proc. 1986 ACM Conf. Lisp and Functional Programming*, pages 249–257, 1986.
- [20] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.
- [21] J. H. Reppy and J. G. Riecke. Simple object for Standard ML. *Proc. 1996 ACM Conf. Programming Language Design and Implementation*, 1996.
- [22] D. A. Schmidt. *Denotational Semantics - A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
- [23] M. Spivey. *Understanding Z*. Cambridge Univ. Press, 1988.
- [24] G. L. Steele, Jr. Building interpreters by composing monads. *Proc. 21st ACM Symp. Principles of Programming Languages*, pages 472–492, 1994.
- [25] C. Weber. Creation of a family of compilers and runtime environments by combining reusable components. *Proc. 4th Int. Conf. Compiler Construction*, pages 110–124, 1992.