An Object-Oriented Denotational Semantics of a Small Programming Language[1]

Jin Song Dong
Software Engineering Group
CSIRO Division of Information Technology, AUSTRALIA

Roger Duke and Gordon Rose
Software Verification Research Centre
The University of Queensland, AUSTRALIA

Abstract

A denotational semantics of a programming language specifies the meaning of the language in terms of well defined mathematical structures. To this end, state-based formal specification notations such as VDM and Z can be used to specify denotational semantics by defining the transition from language constructs into mathematical entities. However, when using such notations, the abstract syntax, static semantics and dynamic semantics are typically defined separately using distinct formal structures. As a consequence, adding a new construct to the language usually requires modification of each of the above structures. Furthermore, this traditional denotational semantic style generally has limited modularity and reusability. This paper advocates the use of the Object-Z notation to give an object-oriented specification of programming language semantics. With this approach, programming language constructs, such as expressions and statements, are viewed as objects, with the result that the abstract syntax, static and dynamic semantics of individual language constructs become encapsulated in one class. This modularity assists readability and facilitates subsequent semantic modification with minimal disruption.

Keywords: denotational semantics, formal methods, object orientation

[1] In Object-Oriented Systems (OOS) journal 4(1):29-52, Chapman Hall, 1997. (ISSN: 09699776)

1 Introduction

Formal description of programming languages supports language standardisation, program verification and software reliability. The state-based formal specification languages VDM [14] and Z [21] have been used to define the denotational semantics of programming languages [2, 13, 22] Indeed, as Jones [15] states,

> 'The main stimulus for the inception of VDM was the description of programming languages.'

The approaches of VDM and Z are similar (see Hayes [13]) for a comparison of the two languages when applied to the specification of the semantics of the small Pascal-like procedural language defined by Jones [15]). In both cases, the definition of a programming language is derived by

- constructing an abstract syntax,
- specifying the consistency constraints (static semantics) that ensure that all language constructs (expressions, statements, etc.) are statically well formed (typically, well typed), and
- specifying the denotations (values) thereby capturing the meaning (dynamic semantics) of the language constructs.

As a consequence of this approach, the abstract syntax, static semantics and dynamic semantics are defined separately and involve the construction of distinct formal structures. This semantic representation is not readily extendible, for if the programming language is enhanced, each of these structures must be extended, usually by extension of case analyses. Mosses [16] has observed the general lack of modularity and reusability of the established denotational semantic approaches. We believe that the main reason for this is that the traditional meta languages used to specify programming language semantics lack a suitable modular structure.

As object-oriented modelling techniques support modularity and reusability, it seems appropriate to apply such techniques to specify programming language semantics. This paper advocates using the Object-Z notation [11, 18] to give an object-oriented specification of programming language semantics. Language constructs such as variables, expressions, statements and procedures are modelled semantically as objects whose attributes and operations capture the role of the construct. A program is specified in terms of its underlying semantic structures (i.e. the objects associated with the language constructs) and their interaction. With this approach, the abstract syntax, static semantics and dynamic semantics of an individual language construct is encapsulated in a single class. Typically, the addition of a new language construct only requires the definition of a new class.

In Jones [15] (1990), VDM is used to specify the semantics of a macro Pascal-like programming language; in Hayes [13], Z is used to specify the semantics of a very similar language, and in the process draws a comparison between the Z and VDM approaches to programming language semantics. This paper uses Object-Z to specify the semantics of a small programming language similar to that defined by Jones [15].

In [8], we briefly presented the idea of using an object-oriented approach to model simple programming language constructs. However, the target programming language modelled in that paper was very simple with no block structuring. Here we not only extend the language model to include block structuring, but also use recently developed object modelling constructs, such as object containment, to create a much more compact model.

It is assumed the reader of this paper has a sound understanding of object orientation, and is familiar with the basic Object-Z notation as described by Duke *et. al.* [11] or Rose [18]. Those less-basic aspects of Object-Z required for defining programming language semantics, such as class union, object containment, operation composition, etc. are discussed in Section 2.

In Section 3, the syntax of a small programming language similar to that given by Jones [15] is defined. The denotational semantics of this language is specified using Object-Z by defining the semantics of expressions, statements and procedures in Sections 4, 5 and 6 respectively, and bringing this together in Section 7 to define the semantics of programs. Notice that the small language whose semantics is specified is not itself an object-oriented language.

2 Aspects of Object-Z

In this section, those specific aspects of Object-Z, such as class union, object containment, operation composition, etc. needed later in this paper to define programming language semantics are introduced and briefly discussed.

2.1 Class Union

Underlying the semantics of an Object-Z specification is the understanding that each object has a unique persistent identity [10, 12] that distinguishes it from every other object, regardless of class. An object's identity is invariant irrespective of changes to its state.

If A is the name of a class, let the identifier A also denote the set of identities of possible objects of class A.[2] Let $\mathbb{O}$ denote the set of all object identities, regardless of class, that may be in the system.

If A and B are distinct classes in the system,

$$A \subseteq \mathbb{O},\ \ B \subseteq \mathbb{O} \text{ and } A \cap B = \varnothing$$

i.e. the identities of objects of class A are distinct from those of class B, but all are elements of $\mathbb{O}$.

Informally, we do not distinguish between an object and its identity, i.e. if we refer to some *object* in $\mathbb{O}$, it is with the understanding that we are in fact referring to the object whose identity is in $\mathbb{O}$. As object identities uniquely identify objects, no ambiguity arises.

Subsets of $\mathbb{O}$ can be formed by the union of sets of object identities, e.g. if A and B are distinct classes, $A \cup B$ is the collection of identities in $\mathbb{O}$ of objects of classes A or B. In this case, if we define $C \mathrel{\widehat{=}} A \cup B$ then C is not itself a class but only a set of object identities. C is called a *class union*.

For example, as will be seen in Section 4, an expression in the small programming language is either a 'constant', a 'variable reference', a 'plus-expression', a 'less-expression', an 'and-expression' or a 'not-expression'. In Object-Z this aspect is captured by the class-union

$$Exp \mathrel{\widehat{=}} Const \cup Varef \cup PlusExp \cup LessExp \cup AndExp \cup NotExp$$

where $Const$, $Varef$, $PlusExp$, $LessExp$, $AndExp$ and $NotExp$ are classes specified in Section 4. With such a class union defined, it now becomes possible to make polymorphic type declarations. For example, $e : Exp$ declares e to be an object of one of the above 6 expression classes; the actual class of e is not specified.

For a detailed discussion of class-union see Dong and Duke [4, 3].

[2] In particular cases, it will be clear from the context whether an identifier is being used as a name of a class or to denote the set of identities of objects of that class.

2.2 Secondary Attributes

Attributes of a class are partitioned into primary and secondary. The values of the primary attributes determine the state of an object; the values of the secondary attributes depend upon the primary attributes of this or other objects in the system and enable an object to retain information about various aspects of the state of the system.

For example, consider classes *Num* and *Plus*:

Num

$z : \mathbb{Z}$

various operations

Plus

$a_1, a_2 : Num$
Δ
sum

$sum = a_1.z + a_2.z$

various operations

The class *Num* has one (primary) attribute, z, of type integer. For the class *Plus*, objects a_1 and a_2 are primary attributes, while sum is a secondary attribute whose value depends upon $a_1.z$ and $a_2.z$. In Object-Z, the declaration of primary and secondary attributes are syntactically separated by the Δ symbol.

Secondary attributes are implicitly included in the Δ-list of every operation in the system. (The understanding of the Δ-list of an operation in Object-Z is that attributes which are so listed are subject to change. Secondary attributes are therefore always subject to change whenever an operation is invoked; primary attributes not included in the Δ-list remain unchanged.) In the above case, if p, say, is an object of class *Plus*, the value of $p.sum$ can change if either of the objects $p.a_1$ or $p.a_2$ undergoes an operation that changes the value of its z attribute, even if p itself undergoes no operation. This idea will be exploited when defining the value of expressions in Section 4.

For a detailed discussion of secondary attributes see Dong *et. al.* [9].

2.3 Referencing Local Objects

Within an object-oriented system, typically objects reference other objects in the system so as to facilitate the sending and receiving of messages. In general, the association between objects determined by the object references will result in a complex structure whose architecture is a crucial part of a system's specification.

When considering a system from the point of view of a particular object, it is often the case that various objects referenced by the given object can be considered as local to the object. For example, the variables, expressions and statements inside a block of a block-structured program are local to the block, a notion that leads directly to the important concept of scope within the program. Furthermore, these local expression and statement structures form an acyclic pattern: inside an expression may be other expressions, but an expression cannot be inside itself; likewise inside a statement may be other statements, but a statement cannot be inside itself.

To capture these ideas within Object-Z, associated with each class is an implicit secondary attribute *dlocal* denoting the subset of $\mathbb{O}$ consisting of all local objects directly referenced by objects of the class. For example, suppose A, B and C are classes and D is defined by

D

$$
\begin{array}{l}
a : A \\
sa : \mathbb{P}\, A \\
b : B \\
sb : \mathbb{P}\, B \\
sc : \operatorname{seq} C
\end{array}
$$

etc.

If the local objects in D are the object a, the objects in the set sb and the objects in the sequence sc, then

$$dlocal = \{a\} \cup sb \cup \operatorname{ran} sc.$$

Hence, if ob is an object of class D, $ob.dlocal$ denotes the set of local objects directly referenced by ob, i.e.

$$ob.dlocal = \{ob.a\} \cup ob.sb \cup \operatorname{ran} ob.sc.$$

If the role of an attribute is to always identify a local object, this can be indicated when the attribute is declared by appending a subscript ‘$_{\textcircled{\scriptsize L}}$’ to the appropriate type. This removes the necessity to write an explicit predicate to define the value of the implicit secondary attribute *dlocal*. For example, adopting this syntactic convention in the specification of the class D gives

$$
\begin{array}{|l}
\multicolumn{1}{l}{D} \\
\hline
\begin{array}{|l}
\hline
a : A_{\copyright} \\
sa : \mathbb{P}\, A \\
b : B \\
sb : \mathbb{P}\, B_{\copyright} \\
sc : \mathrm{seq}\, C_{\copyright} \\
\hline
\end{array} \\
\text{etc.} \\
\hline
\end{array}
$$

Also associated with each class is an implicit secondary attribute *local* denoting the set of local objects directly or indirectly referenced by objects of the class. To be specific, for any object ob,

$$
\begin{aligned}
ob.local = \{obj : \mathbb{O} \mid \exists\, s : \mathrm{seq}_1\, \mathbb{O} \bullet\ & \\
& s(1) \in ob.dlocal \\
& s(\#s) = obj \\
& \forall\, i : 1 \mathrel{..} \#s - 1 \bullet s(i+1) \in s(i).dlocal\}
\end{aligned}
$$

i.e. $obj \in ob.local$ if there is a sequence of objects that starts with ob and ends with obj, and where each object in the sequence is a local object directly referenced by its predecessor object in the sequence. To put this another way, $obj \in ob.local$ if $obj \in ob.dlocal$ or if $obj \in obj_1.local$ for some $obj_1 \in ob.dlocal$.

The acyclic structure of the local object references can now be easily stated:

$$\forall\, ob : \mathbb{O} \bullet ob \notin ob.local.$$

This predicate is an implicit global invariant of any system. When specifying the semantics of the small language later in the paper, this invariant will ensure that expressions and statements do not contain themselves. In addition, the attribute *local* will be used to formulate the requirement that variables and procedures referenced directly or indirectly within a block are all in the scope of the block.

2.4 Object Containment

Consider a system consisting of car and wheel objects. Each car will have attributes that reference its wheels. As wheels are not shared between cars, distinct car objects will reference distinct wheel objects.

This association between cars and their wheels is an example of object containment. A car is said to *contain* its wheels. The basic property of object containment is that an object cannot be contained by two distinct objects at the same time. When formally specifying a system, specifying which objects are contained within other objects gives valuable information about the object reference structure.

To capture this idea, associated with each Object-Z class is an implicit secondary attribute *dcontain* denoting the subset of $\mathbb{O}$ consisting of all objects directly contained by objects of the class. This attribute satisfies two basic conditions:

$$dcontain \subseteq dlocal$$

i.e. a contained object reference is also a local reference, and

$$\begin{array}{l} \forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ \qquad ob_1 \neq ob_2 \;\Rightarrow\; ob_1.dcontain \cap ob_2.dcontain = \varnothing \end{array}$$

i.e. no object is contained by two distinct objects.

Because contained objects are also local objects, combining the acyclic structure of local object references with the second condition above implies that the contained object references determine a tree structure (while local object references form only a directed acyclic graph. Figure 1 illustrates the relationship between local and contained objects.

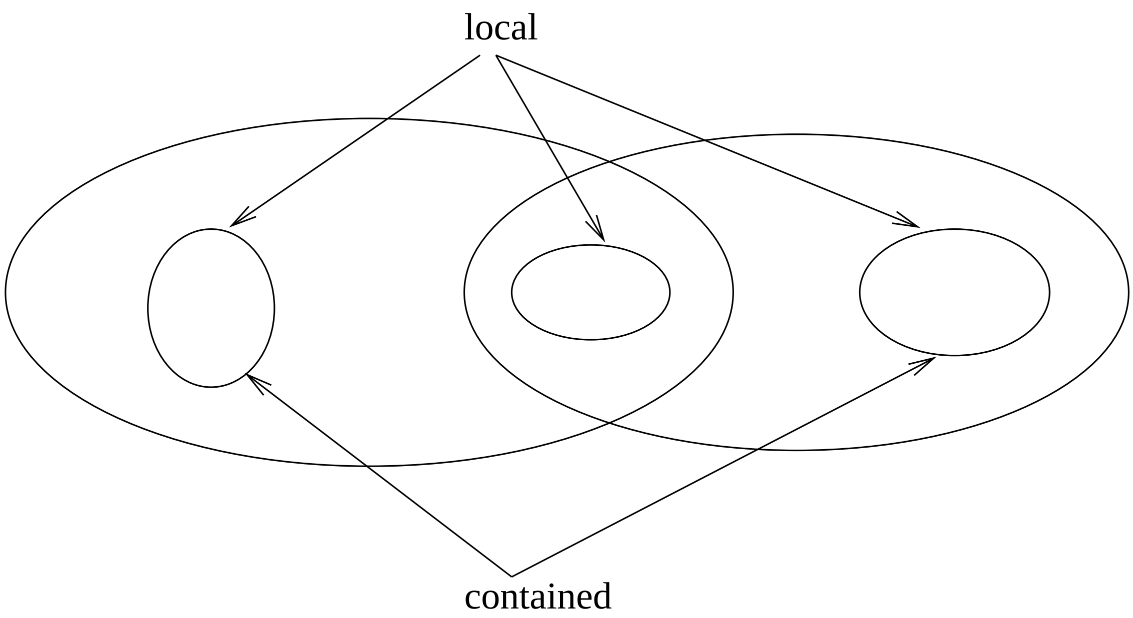

Figure 1: Local and Contained Objects

If the role of an attribute is to always identify a contained object, this can be indicated when the attribute is declared by appending a subscript '$_{©}$' to the appropriate type. For example, suppose A, B and C are classes and D is defined by

D

$$
\begin{array}{|l}
a : A_{\textcircled{L}} \\
sa : \mathbb{P}\, A \\
b : B_{\copyright} \\
sb : \mathbb{P}\, B_{\copyright} \\
sc : \mathrm{seq}\ C_{\textcircled{L}}
\end{array}
$$

etc.

In this case, $dcontained = \{b\} \cup sb$. Notice that $dlocal = \{a, b\} \cup sb \cup \mathrm{ran}\ sc$ because contained references are also local references, i.e. there is an implied subscript '$_{\textcircled{L}}$' associated with every '$_{\copyright}$' subscript.

Later in this paper the notion of object containment is used to capture the block structure of programs: a block *contains* its references to other blocks it declares. Using object containment ensures that the relation between declared blocks forms a tree structure.

For a detailed discussion of object containment see Dong and Duke [6, 5].

2.5 Composing Operations

Four composition operators, conjunction, parallel, choice and sequential composition, will be needed in this paper to define the semantics of the small programming language. To illustrate these operators, consider the following (rather contrived) Object-Z example.

The class *MemLoc* specifies memory location objects.

$MemLoc$

$$n : \mathbb{N}$$

$Write$

$$
\begin{array}{|l}
\Delta(n) \\
n? : \mathbb{N} \\
\hline
n' = n?
\end{array}
$$

$ReadWrite$

$$
\begin{array}{|l}
\Delta(n) \\
n?, n! : \mathbb{N} \\
\hline
n! = n \land n' = n?
\end{array}
$$

A memory-location object has one attribute, n, whose value is a natural number, and two operations, *Write* and *ReadWrite*. The operation *Write* sets the value of n to equal the value of the input $n?$. The operation *ReadWrite* also sets the value of

n to equal $n?$, but in addition it outputs $n!$ whose value equals the original (pre-) value of n. Suppose a system consists of two memory location objects:

System

$a, b : MemLoc$

$a \neq b$

various operations

In this section the composition operators will be used to define various system operations involving interaction between the two memory location objects.

Conjunction

The operation

$$WriteToBoth \mathrel{\widehat=} a.Write \wedge b.Write$$

defines the system operation $WriteToBoth$ whose effect is to perform the operation $Write$ to both of the objects a and b.

The conjunction operator '$\wedge$' in general defines a system operation in terms of two given operations performed concurrently. The input/output parameters of the resulting system operation are the union of the input/output parameters of the component operations; identically named input/output parameters in both components must take the same value. Hence in this case, after the operation both $a.n$ and $b.n$ have the same value, namely the value of the input $n?$.

Parallel

The parallel operator '$\|$' is like conjunction, except that if an output parameter from one component operation has a complementary input parameter in the other component (i.e. both have the same name apart from the ? and ! decorations) these input/output parameters are identified and hidden from the environment, i.e. they are no longer inputs from, or outputs to, the environment. This allows information to be passed internally between objects. Input/output parameters that are not complementary in this sense communicate with the environment as was the case with conjunction.

As an example, consider the operation

$$Swap \mathrel{\widehat=} a.ReadWrite \parallel b.ReadWrite$$

which has the effect that the values of $a.n$ and $b.n$ are swapped. In this case all communication is internal: nothing is input from or output to the environment.

Choice

The choice operator '$\,[\!]\,$' defines a system operation in terms of a choice between two given operations. The input/output parameters of both component operations must be identical, for static typing reasons. The choice is angelic in that if only one component operation is enabled that operation will be selected; if both are enabled the selection is non-deterministic.

For example, the operation

$$WriteToEither \mathrel{\widehat=} a.Write \;[\!]\; b.Write$$

has the effect that non-deterministically one of $a.n$ or $b.n$ is changed to take the value $n?$ while the other is unchanged.

Sequential

The sequential operator '$\fatsemi$' defines a system operation in terms of the sequential composition of two given operations. The first component operation is performed, resulting in some change in the system's state; then the second component operation is performed on this new system state. In addition, if an output parameter from the first component has a complementary input parameter in the second component, a communication channel is set up between those parameters and is hidden from the environment. This is similar to the mechanism of information passing for the parallel operator, except that with the parallel operator information can pass between components in both directions, whereas with the sequential operator information passes only from the first component to the second. Also, with the parallel operator both component operations occur concurrently and act on the same pre-state, whereas with the sequential operator there is an intermediate state after the first component operation has been performed that becomes the pre-state for the second component operation.

For example, the (rather contrived) operation

$$Unchanged \mathrel{\widehat=} a.ReadWrite \fatsemi a.Write$$

has the effect that the value of $a.n$ is unchanged. The first component operation, $a.ReadWrite$, changes the value of $a.n$ to the input $n?$, but the original value of

$a.n$ is communicated to the second component operation, $a.Write$, because the output $n!$ from the first component is complementary to the input $n?$ to the second component and hence they are equated and hidden. Notice that because of this internal communication, the input $n?$ to the second component is not equated to the input $n?$ to the first component.

For a detailed discussion of these composition operators see Smith [20, 19].

3 Syntax of the Small Programming Language

Figure 2 gives the concrete syntax, presented in BNF-like notation, of a small Pascal-like procedural programming language similar to that defined by Jones [15], but in which procedures have both value and reference (var) parameters (so that a distinction can be drawn between the two parameter-passing mechanisms). The scope rules of this language are similar to those in Pascal.

This concrete syntax describes only the external appearance of programs written in the language. In this paper the Object-Z notation will be applied to give an abstract syntax and denotational semantics for the language.

Abbreviations

For uniformity in the definitions throughout this paper, the following abbreviations will be used:

avpl	actual value parameter list	int	integer
arpl	actual reference parameter list	intval	integer value
bool	boolean	loc	location
boolval	boolean value	proc	procedure
const	constant	ref	reference
dec	declaration	stmt	statement
exp	expression	scal	scalar
fvpl	formal value parameter list	val	value
frpl	formal reference parameter list	var	variable

$$
\begin{array}{rcl}
\mathit{Program} & \rightarrow & \text{`Program'}\ \mathit{Id}\ \mathit{Block} \\
\mathit{Id} & \rightarrow & \mathit{Letter} \mid \mathit{Letter}\ \mathit{Id} \\
\mathit{Letter} & \rightarrow & \text{`a'} \mid \text{`b'} \mid \text{`c'} \mid \ldots \mid \text{`z'} \\
\mathit{Block} & \rightarrow & \text{`dec:'}\ \mathit{Declist}\ \mathit{Procedures}\ \text{`begin'}\ \mathit{Stmt}\ \text{`end'} \\
\mathit{Declist} & \rightarrow & \epsilon \mid \mathit{Declaration}\ \mathit{Declist} \\
\mathit{Declaration} & \rightarrow & \mathit{Var}\ \text{`:'}\ \mathit{Type}\ \text{`;'} \\
\mathit{Procedures} & \rightarrow & \epsilon \mid \text{Procedure}\ \mathit{Procedures} \\
\mathit{Procedure} & \rightarrow & \text{`Procedure'}\ \mathit{Id}\ \text{`('}\ \mathit{Declist}\ \text{`var:'}\ \mathit{Declist}\ \text{`);'}\ \mathit{Block} \\
\mathit{Var} & \rightarrow & \mathit{Id} \\
\mathit{Type} & \rightarrow & \text{`integer'} \mid \text{`boolean'} \\
\mathit{Stmt} & \rightarrow & \mathit{AssignStmt} \mid \mathit{IfStmt} \mid \mathit{WhileStmt} \mid \mathit{CallStmt} \mid \mathit{SeqStmt} \\
\mathit{AssignStmt} & \rightarrow & \mathit{Var}\ \text{`:='}\ \mathit{Exp}\ \text{`;'} \\
\mathit{IfStmt} & \rightarrow & \text{`if'}\ \mathit{Exp}\ \text{`then'}\ \mathit{Stmt}\ \text{`else'}\ \mathit{Stmt} \\
\mathit{WhileStmt} & \rightarrow & \text{`while'}\ \mathit{Exp}\ \text{`do'}\ \mathit{Stmt} \\
\mathit{CallStmt} & \rightarrow & \mathit{Id}\ \text{`('}\ \mathit{Explist}\ \mathit{Varlist}\ \text{`);'} \\
\mathit{Exp} & \rightarrow & \mathit{BoolConst} \\
 & & \mid \mathit{IntConst} \\
 & & \mid \mathit{Var} \\
 & & \mid \mathit{Exp}\ \text{`}{+}\text{'}\ \mathit{Exp} \\
 & & \mid \mathit{Exp}\ \text{`}{<}\text{'}\ \mathit{Exp} \\
 & & \mid \mathit{Exp}\ \text{`}\wedge\text{'}\ \mathit{Exp} \\
 & & \mid \text{`}\neg\text{ '}\ \mathit{Exp} \\
\mathit{BoolConst} & \rightarrow & \text{`true'} \mid \text{`false'} \\
\mathit{IntConst} & \rightarrow & \mathit{Digit} \mid \mathit{IntConst}\ \mathit{Digit} \\
\mathit{Digit} & \rightarrow & \text{`0'} \mid \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'} \\
\mathit{Explist} & \rightarrow & \epsilon \mid \mathit{Exp}\ \text{`,'}\ \mathit{Explist} \\
\mathit{Varlist} & \rightarrow & \epsilon \mid \mathit{Var}\ \text{`,'}\ \mathit{Varlist} \\
\mathit{SeqStmt} & \rightarrow & \text{`begin'}\ \mathit{Stmtseq}\ \text{`end'} \\
\mathit{Stmtseq} & \rightarrow & \epsilon \mid \mathit{Stmt}\ \mathit{Stmtseq}
\end{array}
$$

Figure 2: Concrete Syntax of the Small Programming Language

4 Expressions

In this section, Object-Z is used to specify the abstract syntax and denotational semantics of expressions. Each expression is modelled as an object: a binary (unary) expression is modelled as an object with two (one) sub-expression local object attributes, and with a value derived from the values of these sub-expressions. An expression is either a 'constant', a 'variable reference', a 'plus-expression', a 'less-expression', an 'and-expression' or a 'not-expression', according to the concrete syntax given in Figure 2. In Object-Z this aspect is captured by the class-union

$$Exp \mathrel{\widehat=} Const \cup Varef \cup PlusExp \cup LessExp \cup AndExp \cup NotExp$$

where the classes $Const$, $Varef$, $PlusExp$, $LessExp$, $AndExp$ and $NotExp$ are each specified in this section.

To begin, as the scalar type of an expression is boolean or integer, define

$$Scaltype ::= bool \mid int.$$

Likewise, the value of an expression is either boolean or integer, and this can be expressed, using the Z free-type construct [1, 21][3], by

$$Scalval ::= boolval\langle\langle \mathbb{B} \rangle\rangle \mid intval\langle\langle \mathbb{Z} \rangle\rangle.$$

The function $type\_of$ captures the relation between expression values and types:

$$\begin{array}{|l}
type_of : Scalval \rightarrow Scaltype \\
\hline
\forall\, b : \mathbb{B};\ n : \mathbb{Z} \bullet \\
\qquad type_of(boolval(b)) = bool \\
\qquad type_of(intval(n)) = int
\end{array}$$

Constants

A constant is an object of the class $Const$.

$$\begin{array}{|l}
\underline{\ Const\ } \\
\begin{array}{|l}
\hline
type : Scaltype \\
val : Scalval \\
\hline
type_of(val) = type \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\underline{\ OutVal\ } \\
val! : Scalval \\
\hline
val! = val \\
\hline
\end{array} \\
\hline
\end{array}$$

[3]A comparison between free-type in Z and class-union in Object-Z is given in [3].

The type of a constant is represented by attribute *type*, and the meaning (value) by attribute *val*. The class invariant ensures that the type of the constant's value is consistent with the constant's type. The operation *OutVal* enables a constant to broadcast (output) its value; this operation will be common to all expressions.

Variable References

In order to specify variable references, first the concept of variable locations, modelled by the class *Varloc* which inherits the class *Const*, is needed to capture the notion of a memory store in which values of variables can be stored.

$$
\begin{array}{|l}
\textit{Varloc} \\
\hline
\textit{Const} \\
\\
\begin{array}{|l}
\textit{Assign} \\
\hline
\Delta(\textit{val}) \\
\textit{val}? : \textit{Scalval} \\
\hline
\textit{val}' = \textit{val}? \\
\hline
\end{array} \\
\hline
\end{array}
$$

Notice that although *Varloc* inherits *Const*, variable locations differ from constants in that the attribute *val* can be changed (by the operation *Assign*).

Variable references are specified by the class *Varef* containing attributes *id* denoting the variable's name (identifier), and *varloc* denoting a reference to the variable location object where the current value of the variable is stored. $[Id]$ denotes the set of all possible identifiers.

$$
\begin{array}{|l}
\textit{Varef} \\
\hline
\begin{array}{|l}
\hline
\textit{id} : \textit{Id} \\
\textit{varloc} : \textit{Varloc} \\
\Delta \\
\textit{type} : \textit{Scaltype} \\
\textit{val} : \textit{Scalval} \\
\hline
\textit{type} = \textit{varloc}.\textit{type} \\
\textit{val} = \textit{varloc}.\textit{val} \\
\hline
\end{array}
\quad
\begin{array}{l}
\begin{array}{|l}
\textit{OutLoc} \\
\hline
\textit{varloc}! : \textit{Varloc} \\
\hline
\textit{varloc}! = \textit{varloc} \\
\hline
\end{array} \\
\begin{array}{|l}
\textit{ReLoc} \\
\hline
\Delta(\textit{varloc}) \\
\textit{varloc}? : \textit{Varloc} \\
\hline
\textit{varloc}' = \textit{varloc}? \\
\hline
\end{array}
\end{array} \\
\\
\textit{OutVal} \mathrel{\widehat=} \textit{varloc}.\textit{OutVal} \\
\textit{Assign} \mathrel{\widehat=} \textit{varloc}.\textit{Assign} \\
\hline
\end{array}
$$

The attribute *varloc* in this class is not a contained object reference, allowing different variable references to have the same variable location. This property, together with the operations *OutLoc* and *ReLoc*, will enable the specification of reference parameter substitution in procedure calls in Section 6.

Binary and Unary Expressions

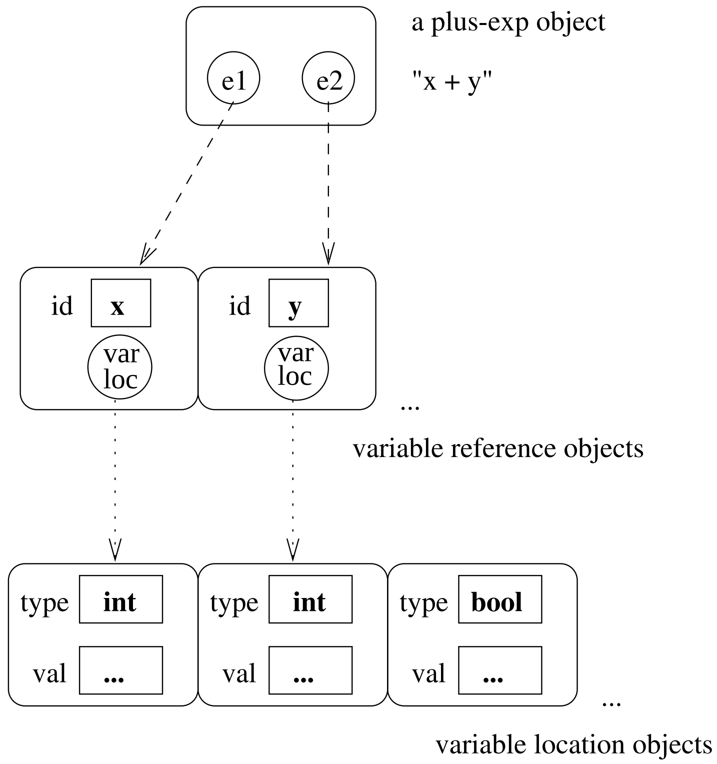

Figure 3: Reference Structure of a Plus Expression

Classes *PlusExp*, *LessExp* and *AndExp* of specific binary expression objects are defined by inheriting the generic class *BinaryExp*. Similarly, the class *NotExp* of specific unary expression objects is defined by inheriting *UnaryExp*. The meaning (value) of a specific expression object is dependent upon the meaning of its component sub-expressions.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BinaryExp}} \\
\hline
\begin{array}{|l}
\hline
e_1, e_2 : Exp_{\copyright} \\
type : Scaltype \\
\Delta \\
val : Scalval \\
\hline
type\_of(val) = type \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OutVal}} \\
\hline
val! : Scalval \\
\hline
val! = val \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{UnaryExp}} \\
\hline
\begin{array}{|l}
\hline
e : Exp_{\copyright} \\
type : Scaltype \\
\Delta \\
val : Scalval \\
\hline
type\_of(val) = type \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OutVal}} \\
\hline
val! : Scalval \\
\hline
val! = val \\
\hline
\end{array} \\
\hline
\end{array}
$$

The subscript '$\copyright$' on the type *Exp* specifies that e_1, e_2 and e are local references, ensuring that the reference structure of expressions is acyclic. Figure 3 shows the object reference structure of a plus expression.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PlusExp}} \\
\hline
BinaryExp \\
\begin{array}{|l}
\hline
e_1.type = e_2.type = int \\
type = int \\
val = intval(intval^{\sim}(e_1.val) \\
\qquad\qquad + intval^{\sim}(e_2.val)) \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{LessExp}} \\
\hline
BinaryExp \\
\begin{array}{|l}
\hline
e_1.type = e_2.type = int \\
type = bool \\
val = boolval(intval^{\sim}(e_1.val) \\
\qquad\qquad < intval^{\sim}(e_2.val)) \\
\hline
\end{array} \\
\hline
\end{array}
$$

Note that the super-script '$\sim$' denotes function inverse.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AndExp}} \\
\hline
BinaryExp \\
\begin{array}{|l}
\hline
e_1.type = e_2.type = bool \\
type = bool \\
val = boolval(boolval^{\sim}(e_1.val) \\
\qquad\qquad \wedge boolval^{\sim}(e_2.val)) \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NotExp}} \\
\hline
UnaryExp \\
\begin{array}{|l}
\hline
e.type = bool \\
type = bool \\
val = \\
\quad boolval(\neg\, boolval^{\sim}(e.val)) \\
\hline
\end{array} \\
\hline
\end{array}
$$

The secondary attribute *val* captures the way the value of an expression depends on the value of its sub-expressions; i.e. *val* captures the meaning (dynamic semantics) of an expression.

In this specification, the abstract syntax, static semantics and dynamic semantics for each kind of expression are specified in the one Object-Z class. This compares with the traditional approach using VDM [15] or Z [13], where these three aspects are usually defined using distinct formal structures. For a detailed comparison see Dong *et. al.* [8].

The Ease of Extendibility

A virtue of the object-oriented approach is that if we want to add binary constructions such as an 'or-expression' and a 'multiplication-expression', it is only necessary to add the classes

$$
\begin{array}{l}
\underline{\; OrExp \;} \\
BinaryExp \\
\hline
e_1.type = e_2.type = bool \\
type = bool \\
val = boolval(boolval^{\sim}(e_1.val) \\
\qquad\qquad \vee\ boolval^{\sim}(e_2.val)) \\
\hline
\end{array}
\qquad
\begin{array}{l}
\underline{\; MultExp \;} \\
BinaryExp \\
\hline
e_1.type = e_2.type = int \\
type = int \\
val = intval(intval^{\sim}(e_1.val) \\
\qquad\qquad *\ intval^{\sim}(e_2.val)) \\
\hline
\end{array}
$$

and extend the class-union *Exp* to include the classes *OrExp* and *MultExp*.[4]

This demonstrates the extendibility of the object-oriented approach in defining programming language semantics.

5 Statements

In the last section, each expression was modelled as an object. Similarly in this section, each statement, regardless of its kind, is also modelled as an object that captures the static and dynamic semantics of the statement.

Following the concrete syntax in Figure 2, a statement is either an 'assignment-statement', an 'if-statement', a 'while-statement', a 'sequence-of-statements' or a 'call-statement'. This is captured by the class-union

$$Stmt \mathrel{\widehat=} AssignStmt \cup IfStmt \cup WhileStmt \cup SeqStmt \cup CallStmt$$

where *AssignStmt*, *IfStmt*, *WhileStmt*, *SeqStmt* and *CallStmt* are classes that are specified in this section.

The meaning of a statement, unlike an expression, is not a single value but rather a store transformation. With the traditional approach using VDM or Z, the meaning of an assignment is represented by overridding the old store to create a new store. In the Object-Z representation presented here, such a transformation is effected by assigning a value to a variable.

[4] A comparison of extendibility between this object-oriented approach and the traditional approaches using VDM or Z is given in Dong *et. al.* [8].

Assignment-statements

The class *AssignStmt* modelling an assignment-statement has an attribute *lhs* identifying the variable reference object denoting the variable on the left hand side of the assignment, and an attribute *rhs* identifying the expression object denoting the expression on the right hand side of the assignment.

$$
\begin{array}{|l}
\hline
\textit{AssignStmt} \\
\begin{array}{|l}
\hline
lhs : Varef_{©} \\
rhs : Exp_{©} \\
\hline
lhs.type = rhs.type \\
\hline
\end{array} \\
Execute \mathrel{\widehat=} \\
\quad rhs.OutVal \parallel lhs.Assign \\
\hline
\end{array}
$$

Notice that in the definition of this class, the abstract syntax of an assignment-statement is captured by the attributes *lhs* and *rhs*, the static semantics is captured by the state invariant which requires both sides of the assignment be of the same type, and the dynamic semantics is captured by the operation *Execute* which assigns the value of the expression *rhs* to the variable reference *lhs*. Hence the abstract syntax and the static and dynamic semantics of an assignment-statement are captured by the one class construct. As will be seen, a similar pattern occurs for each of the different statement classes.

The object reference structure of an assignment-statement is illustrated in Figure 4. In this case there are two variable references with the same identifier x. As the figure shows, these are modelled as aliases, a view that will be exploited when specifying procedures in Section 6.

If-statements

An if-statement is modelled by the class *IfStmt* which has an attribute *test* identifying the boolean expression denoting the test part of the statement, and attributes *then* and *else* identifying the sub-statements denoting the alternative branches.

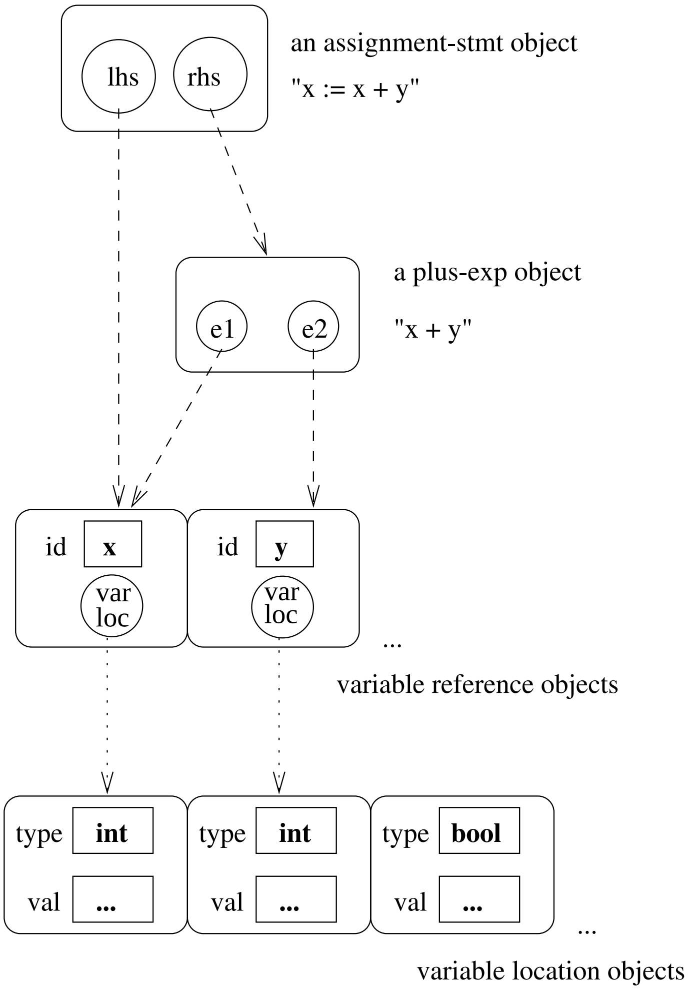

Figure 4: Reference Structure of an Assignment Statement

IfStmt

$test : Exp_{\textcircled{L}}$
$then, else : Stmt_{\textcircled{L}}$

$test.type = bool$

$$
\begin{aligned}
Execute \mathrel{\widehat{=}}\ & [boolval^{\sim}(test.val)] \wedge then.Execute \\
& [\,] \\
& [\neg\ boolval^{\sim}(test.val)] \wedge else.Execute
\end{aligned}
$$

The pre-conditions for each of the two alternatives of *Execute* are disjoint, so that precisely one will be performed deterministically.

While-statements

A while-statement consists of a boolean test expression and a body statement; it is modelled as an object of the class *WhileStmt*.

$$
\begin{array}{l}
\underline{\quad WhileStmt \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
\quad \begin{array}{|l}
\hline
test : Exp_{\copyright} \\
body : Stmt_{\copyright} \\
\hline
test.type = bool \\
\hline
\end{array} \\
\quad Execute \mathrel{\widehat=} ([boolval^{\sim}(test.val)] \wedge body.Execute \mathbin{;} Execute \\
\qquad\qquad\qquad [] \\
\qquad\qquad\qquad [\neg\ boolval^{\sim}(test.val)]
\end{array}
$$

The operation *Execute* is defined recursively using the sequential composition operator. The understanding here is that the operation is only enabled if the recursion terminates, i.e. a postcondition of this operation is that $\neg\ boolval^{\sim}(test.val)$. In Object-Z (as in Z), an operation is enabled only if it satisfies both its pre- and post-conditions.

The object reference structure of a while-statement is illustrated in Figure 5.

Sequence-of-statements

The meaning of a sequence-of-statements is the sequential execution of its component sub-statements; it is modelled as an object of the class *SeqStmt*.

$$
\begin{array}{l}
\underline{\quad SeqStmt \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
\quad \begin{array}{|l}
\hline
stmts : \mathrm{seq}\, Stmt_{\copyright} \\
\hline
\end{array} \\
\quad Execute \mathrel{\widehat=} \mathbin{;}\ i : \mathrm{dom}\, stmts \bullet stmts(i).Execute
\end{array}
$$

The operation *Execute* is defined by using the distributed version of the sequential composition operator; in expanded form it becomes

$$stmts(1).Execute \mathbin{;} stmts(2).Execute \mathbin{;} \ldots \mathbin{;} stmts(\#stmts).Execute.$$

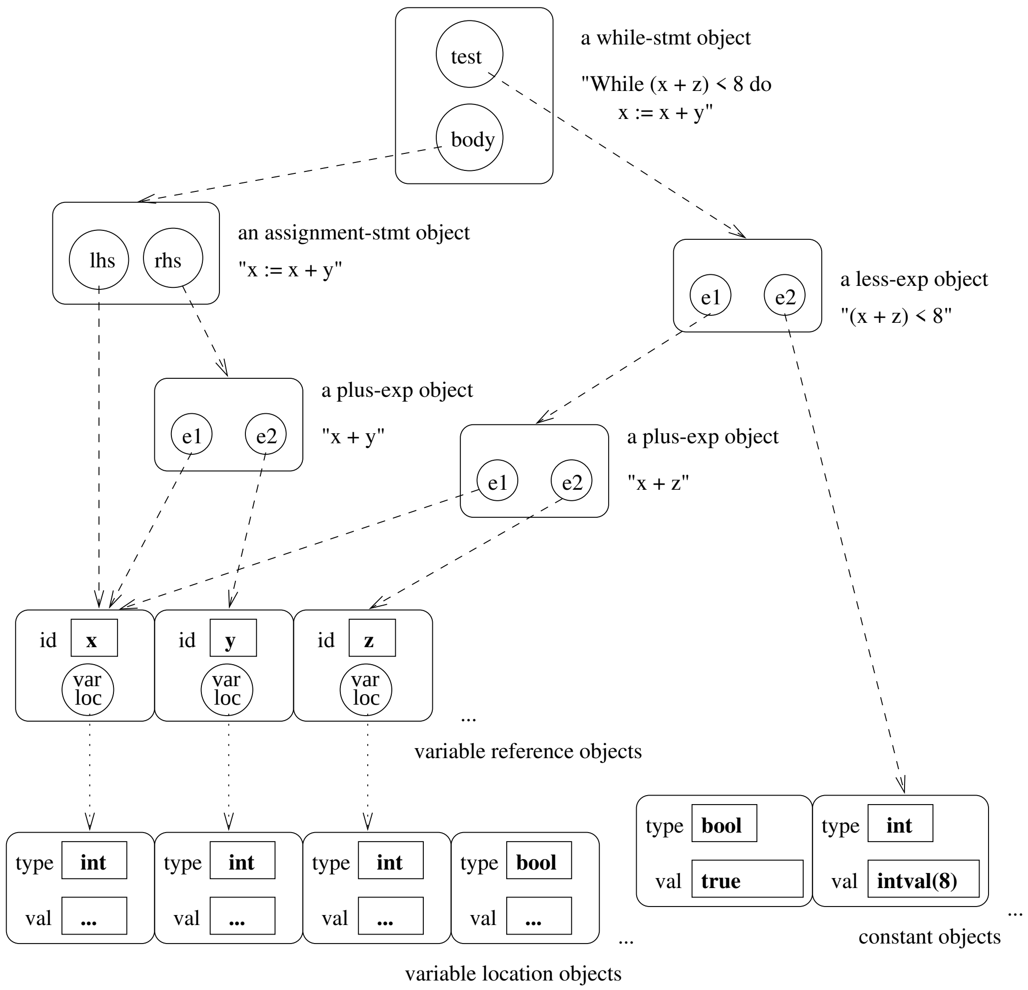

Figure 5: Reference Structure of a *While* Statement

Call-statements

In order to specify call-statements, first the concept of procedure references, modelled by the class *Procref*, is needed.

Procref

$id : Id$
$proc : Procedure$

$Execute \mathrel{\widehat{=}} proc.Execute$

A procedure reference is similar to a variable reference. The attribute *id* denotes the name of the procedure *proc* being referenced. The class *Procedure* is specified in Section 6.

A call-statement is modelled by the class *CallStmt* which has an attribute *procref* identifying a procedure reference that in turn identifies the procedure being called by the statement, an attribute *avpl* identifying the list of expression objects denoting the actual value parameter list, and an attribute *arpl* identifying the list of variable reference objects denoting the actual reference parameter list.

CallStmt

$$
\begin{array}{l}
procref : Procref_{©} \\
avpl : \mathrm{seq}\ Exp_{©} \\
arpl : \mathrm{seq}\ Varef_{©} \\
\hline
\#avpl = \#procref.proc.fvpl \\
\#arpl = \#procref.proc.frpl \\
\forall\, i : \mathrm{dom}\ avpl \bullet avpl(i).type = procref.proc.fvpl(i).type \\
\forall\, i : \mathrm{dom}\ arpl \bullet arpl(i).type = procref.proc.frpl(i).type
\end{array}
$$

$$
\begin{array}{l}
ValSubs \mathrel{\widehat=} \bigwedge i : \mathrm{dom}\ avpl \bullet \\
\qquad\qquad avpl(i).OutVal \parallel procref.proc.fvpl(i).Assign \\
RefSubs \mathrel{\widehat=} \bigwedge i : \mathrm{dom}\ arpl \bullet \\
\qquad\qquad arpl(i).OutLoc \parallel procref.proc.frpl(i).ReLoc \\
Execute \mathrel{\widehat=} (ValSubs \land RefSubs) \fatsemi procref.Execute
\end{array}
$$

The first two predicates of the state schema specify that the length of each actual parameter list of a call statement equals the length of the corresponding formal parameter list of the called procedure. The last two predicates of the state schema specify that the type of each actual parameter matches the type of the corresponding formal parameter. (Formal parameters are specified in Section 6 when defining the class *Procedure*.)

The operation *ValSubs* takes the value of each expression in the actual value parameter list and assigns this value to the variable reference in the corresponding position in the formal value parameter list. The operation *RefSubs*, on the other hand, takes the store location of each variable reference in the actual reference parameter list and relocates the variable reference in the corresponding position in the formal reference parameter list to point to this store position. In other words, with the operation *ValSubs* the values of the actual parameters are internally passed between the actual and formal parameter objects, while with the operation *RefSubs* references to the variable location object of the actual parameters are passed. This distinction is clearly and succinctly made in the Object-Z representation.

The dynamic semantics of a call-statement is specified by the operation *Execute* which is just the execution of the called procedure (i.e. *procref.Execute*) after the parameter substitutions (*ValSubs* and *RefSubs*).

The Ease of Reusability

The reuse of existing classes to define new classes is readily achieved using inheritance. For instance, a repeat-statement can be modelled by the class *RepeatStmt* which inherits the class *WhileStmt* but with the operation *Execute* redefined.

$$
\begin{array}{|l}
\hline
\textit{RepeatStmt} \\
\textit{WhileStmt}[\mathbf{redef:}\ \textit{Execute}] \\
\\
\textit{Execute} \mathrel{\widehat=} \textit{body}.\textit{Execute} \mathbin{;} \\
\qquad\qquad ([\neg\ \textit{boolval}^{\sim}(\textit{test}.\textit{val})] \wedge \textit{Execute} \\
\qquad\qquad [] \\
\qquad\qquad [\textit{boolval}^{\sim}(\textit{test}.\textit{val})]) \\
\hline
\end{array}
$$

This demonstrates the reusability of the object-oriented approach in defining programming language semantics.

6 Procedures

Just as with expressions and statements, a procedure is modelled as an object. As procedures are similar in structure to programs (see Section 7), the common features are specified by an auxiliary class *BlockBase* which is inherited by the classes *Procedure* and *Program*.

$$
\begin{array}{|l}
\hline
\textit{BlockBase} \\
\quad \begin{array}{|ll}
\hline
\textit{localvardecs} : \mathbb{P}\ \textit{Varef} & \\
\textit{localprocdecs} : \mathbb{P}\ \textit{Procrefs} & \\
\textit{stmt} : \textit{Stmt} & \\
\Delta & \\
\textit{staticvarloc} : \textit{Id} \rightarrowtail\!\!\!\!\!\mapsto \textit{Varloc}_{\copyright} & \\
\textit{staticproc} : \textit{Id} \rightarrowtail\!\!\!\!\!\mapsto \textit{Procedure}_{\copyright} & \\
\textit{newvardecs} : \mathbb{P}\ \textit{Varef} & \\
\textit{varscope} : \mathbb{P}\ \textit{Varef} & \\
\textit{procscope} : \mathbb{P}\ \textit{Procref} & \\
\hline
\forall v_1, v_2 : \textit{newvardecs} \bullet v_1 \neq v_2 \;\Rightarrow\; v_1.\textit{id} \neq v_2.\textit{id} & [\mathsf{i}] \\
\textit{Varef} \cap \textit{stmt}.\textit{local} \subseteq \textit{varscope} & [\mathsf{ii}] \\
\forall p : \textit{localprocdecs} \bullet p.\textit{parent} = \textit{self} & [\mathsf{iii}] \\
\forall p_1, p_2 : \textit{localprocdecs} \bullet p_1 \neq p_2 \;\Rightarrow\; p_1.\textit{id} \neq p_2.\textit{id} & [\mathsf{iv}] \\
\textit{Procref} \cap \textit{stmt}.\textit{local} \subseteq \textit{procscope} & [\mathsf{v}] \\
\textit{staticproc} = \{p : \textit{localprocdecs} \bullet p.\textit{id} \mapsto p.\textit{proc}\} & [\mathsf{vi}] \\
\hline
\end{array} \\
\textit{Execute} \mathrel{\widehat=} \textit{stmt}.\textit{Execute} \\
\hline
\end{array}
$$

The features of this class are most readily understood in terms of how the class is inherited; hence the informal explanation is delayed until after the class *Procedure* has been specified. Define the class-union

$$Block \mathrel{\widehat{=}} Procedure \cup Program$$

where the class *Procedure* is specified below and the class *Program* is specified in Section 7.

$$
\begin{array}{|l}
\textit{Procedure} \\
\hline
\textit{BlockBase} \\
\begin{array}{|l}
\textit{fvpl} : \operatorname{seq} \textit{Varef} \\
\textit{frpl} : \operatorname{seq} \textit{Varef} \\
\textit{parent} : \textit{Block} \\
\hline
\langle \textit{localvardecs}, \operatorname{ran} \textit{fvpl}, \operatorname{ran} \textit{frpl} \rangle \text{ partitions } \textit{newvardecs} \qquad [\mathsf{vii}] \\
\textit{varscope} = \textit{parent}.\textit{varscope} \oplus_v \textit{newvardecs} \qquad [\mathsf{viii}] \\
\textit{staticvarloc} = \{ v : (\textit{localvardecs} \cup \operatorname{ran} \textit{fvpl}) \bullet \\
\qquad\qquad v.\textit{id} \mapsto v.\textit{varloc} \} \qquad [\mathsf{ix}] \\
\textit{procscope} = \textit{parent}.\textit{procscope} \oplus_p \textit{localprocdecs} \qquad [\mathsf{x}]
\end{array}
\end{array}
$$

Consider these classes in detail. The attribute *localvardecs* denotes the set of variables declared in a block. Each such declaration is modelled as a variable reference object (with attributes that give the variable's name and the location where the variable's current value is stored—see Section 4). In the case of a procedure, the declarations *localvardecs*, together with the declarations in the sequences *fvpl* and *frpl* denoting the variable and reference formal parameter lists respectively, determine the set *newvardecs* of all new variable declarations (predicate [vii]). By predicate [i] the identifiers of the local variables are distinct.

The secondary attribute *varscope* denotes the set of all variables that are in scope. Again these are modelled as variable reference objects. For a procedure, this set is formed by having the local variables override the set of variables inherited from the parent block (predicate [viii]), where the override operator '$\oplus_v$' is defined by

$$
\begin{array}{|l}
\_ \oplus_v \_ : \mathbb{P}\, \textit{Varef} \times \mathbb{P}\, \textit{Varef} \to \mathbb{P}\, \textit{Varef} \\
\hline
\forall s_1, s_2 : \mathbb{P}\, \textit{Varef} \bullet \\
\qquad s_1 \oplus_v s_2 = (s_1 \setminus \{ v_1 : s_1 \mid \exists v_2 : s_2 \bullet v_1.\textit{id} = v_2.\textit{id} \}) \cup s_2
\end{array}
$$

The parent procedure is denoted by the attribute *parent* and can be either another procedure or the program itself.

Predicate [ii] ensures that every variable reference appearing in the block's statement will be one of the variable references in scope. (Recall that *stmt.local* denotes the set of local objects directly or indirectly referenced by the object *stmt*—see Section 2.3.) Because of predicate [i], variable instances denoted by the same identifier will be modelled by the same variable reference.

The secondary attribute *staticvarloc* is a one-one function that records the variable locations associated with the variable declarations in *localvardecs* and (in the case of a procedure) in the formal variable parameter list *fvpl* (predicate [ix]). The subscript '$_{©}$' ensures that the locations are unique to each block, i.e. that the locations of variables declared in one block are distinct from those of variables declared in any other block. Effectively, these variable locations can be allocated statically and remain unchanged as the program is executed. Notice that the declarations in the formal reference parameter list *frpl* are not included here. As was seen in Section 5 with the specification of call-statements, the locations associated with these variables are allocated dynamically.

The attribute *localprocdecs* denotes the set of procedures declared in the block. As with variable declarations, each procedure declaration is modelled as a procedure reference object (with attributes that give the procedure's name and identify the associated procedure object). The parent of each procedure is the block in which it is declared (predicate [iii]). By predicate [iv] the identifiers of the declared procedures are distinct.

The secondary attribute *staticproc* is a one-one function that explicitly associates a name (identifier) to each procedure declared in the block (predicate [vi]). The subscript '$_{©}$' ensures that the procedures declared in one block are distinct from those declared in any other block.

The secondary attribute *procscope* denotes the set of all procedures that are in scope. Again these are modelled as procedure reference objects. For a block that is a procedure, this set is obtained by having the locally declared procedures override the set of procedures inherited from the parent block (predicate [x]), where the override operator '$\oplus_p$' is defined by

$$\_ \oplus_p \_ : \mathbb{P}\,Procref \times \mathbb{P}\,Procref \rightarrow \mathbb{P}\,Procref$$

$$\forall\, s_1, s_2 : \mathbb{P}\,Procref \bullet$$
$$s_1 \oplus_p s_2 = (s_1 \setminus \{p_1 : s_1 \mid \exists\, p_2 : s_2 \bullet p_1.id = p_2.id\}) \cup s_2$$

Predicate [v] ensures that every procedure reference appearing in the block's statement will reference one of the procedures in scope.

In summary, the whole access scope of a procedure is defined as the scope of the parent of the procedure overridden by the local variable and procedure declarations.

The variables *varscope* and *procscope* in a block essentially play the role of passing the static and dynamic environment from the parent to the child procedure. (Compare this with how the scope rules are specified in the VDM or Z specifications [15, 13].)

The scope rules captured here are similar to those for Pascal, namely that within the statement part of a procedure it is permitted to access all the variables and procedures which are declared locally in the procedure. In addition, access is permitted to all those variables and procedures which can be accessed by the procedure's parent provided they have not been overriden by the procedure's local declarations.

7 Programs

The class *Program* modelling a program in the small language can now be specified. It is defined by inheriting the class *BlockBase* and adding the attribute *id* denoting a program's name.

Program

BlockBase

$id : Id$

$$
\begin{aligned}
&newvardecs = localvardecs\\
&varscope = newvardecs\\
&staticvarloc = \{v : localvardecs \bullet v.id \mapsto v.varloc\}\\
&procscope = procdec
\end{aligned}
$$

This class is similar to the class *Procedure* but with no parameter lists and with no variables or procedures inherited from a parent block. In this class

- the abstract syntax of a program is captured by the attributes *localvardecs* (variable declarations), *localprocdecs* (sub-procedure declarations) and *stmt* (the body of the program);
- the static semantics is captured by the class invariant; and
- the dynamic semantics is captured by the operation *Execute*.

Hence again the different semantics features of the programming language construct are encapsulated within the one object-Z class.

8 Conclusion

Although the lack of modularity and extendibility of traditional semantic representations has been addressed by some researchers (e.g. the work by Mosses on Action Semantics [17], among others), the meta languages used still have limited modular structure. A consequence is that programming language constructs are modelled as semantic functions, not as objects. In this paper an object-oriented approach to the specification of programming language semantics has been adopted. The key idea introduced is to model language constructs such as expressions, variables, statements, procedures, etc. as objects. These objects belong to classes that have their own signature and operations to capture the abstract syntax and semantics of the corresponding language constructs. Using Object-Z to specify a programming language not only leads to a structural and concise specification, but more importantly

- the abstract syntax and static and dynamic semantics can be grouped together for each component of the language structure, and
- a new construct can be readily added to the representation, typically by the addition of a new class, i.e. the representation is extendible.

This paper clearly demonstrates the power of object orientation in expressing denotational semantics. The approach has also been applied successfully to specify the denotational semantics of a simple object-oriented programming language [7]. The authors believe that there would be considerable advantages in using the object-oriented techniques introduced in this paper to specify widely-used programming languages such as Modula3, Eiffel and C++.

References

[1] R. Arthan. On free type definitions in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, pages 40–58, University of York, Dec 1991.

[2] D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* International Series in Computer Science. Prentice-Hall, 1982.

[3] J.S. Dong. Living with Free Type and Class Union. In *The 1995 Asia-Pacific Software Engineering Conference (APSEC'95)*, pages 304–312. IEEE Computer Society Press, December 1995.

[4] J.S. Dong and R. Duke. Class Union and Polymorphism. In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Proc. 12th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 12 & 9*, pages 181–190. Prentice-Hall, November 1993.

[5] J.S. Dong and R. Duke. Exclusive Control within Object Oriented Systems. In *The 18th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'Pacific95)*, pages 123–132. Prentice-Hall, November 1995.

[6] J.S. Dong and R. Duke. The Geometry of Object Containment. *Object-Oriented Systems*, 2(1):41–63, Chapman & Hall, March 1995.

[7] J.S. Dong and R. Duke. A formal object model of an object-oriented programming language. In *The 20th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'USA)*, Santa Barbara, USA, 1996.

[8] J.S. Dong, R. Duke, and G. Rose. An Object-Oriented Approach to the Semantics of Programming Languages. In G. Gupta, editor, *Proc. 17th Annual Computer Science Conference (ACSC'17)*, pages 767–775, NZ, January 1994.

[9] J.S. Dong, G. Rose, and R. Duke. The Role of Secondary Attributes in Formal Object Modelling. In Alex Stoyenko, editor, *The First IEEE International Conference on Engineering Complex Computer Systems (ICECCS'95)*, pages 31–38, Ft. Lauderdale, USA, November 1995. IEEE Computer Society Press.

[10] R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.

[11] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[12] A. Griffiths and G. Rose. A Semantic Foundation for Object Identity in Formal Specification. *Object-Oriented Systems*, 2:195–215, Chapman & Hall 1995.

[13] I. Hayes. A small language definition in Z. Technical Report 94-50, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1994.

[14] C. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice-Hall, 1986.

[15] C. B. Jones. A small language definition. In C. B. Jones and R. Shaw, editors, *Case Studies in Systematic Software Development*, International Series in Computer Science. Prentice-Hall, 1990.

[16] P. D. Mosses. Denotional semantics. In *Handbook of Theoretical Computer Science*, volume B: Formal Model and Semantics, pages 577–631. MIT Press, 1990.

[17] P. D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[18] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[19] G. Smith. A Logic for Object-Z (Additional Rules). Technical Report 95-26, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1995.

[20] G. Smith. Reasoning about Object-Z specifications. In *Proc. Asia Pacific Software Engineering Conference '95 (APSEC'95)*, pages 489–497. IEEE Computer Society Press, 1995.

[21] J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 2nd edition, 1992.

[22] S. Stepney. *High integrity compilation: A case study.* Prentice-Hall, 1993.