An Object-Oriented Approach to the Semantics of Programming Languages

Jin Song Dong **Roger Duke** **Gordon Rose**

Software Verification Research Centre
Department of Computer Science
University of Queensland, Australia

jason@cs.uq.oz.au rduke@cs.uq.oz.au rose@cs.uq.oz.au

Abstract

State-based formal specification languages such as VDM and Z have been used to define the semantics of programming languages. Usually, the abstract syntax, static semantics and dynamic semantics of the programming language are defined separately and involve the construction of distinct formal structures. However, if the programming language is enhanced, extending the semantics may require modifications to each of the above structures. In this paper we present an object-oriented formal definition of the semantics of a small procedural programming language using the Object-Z specification language. With this approach, the abstract syntax, static semantics and dynamic semantics of an individual construct are typically defined in one class. Not only does this help the readability of the semantics, but if the language is enhanced the corresponding semantic modifications can be captured by minimal disruption of the existing semantics. Examples show the addition of new kinds of expression by corresponding class definitions[1].

1 Introduction

The state-based specification languages VDM and Z can be used to define the semantics of programming languages[2, 4, 7, 8]. Indeed, as Jones[8] states, the

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

[1]In Proceedings of the *17th Annual Computer Science Conference (ACSC'17)*, pages 767-775, Jan 1994

The approaches of VDM and Z are very similar (see [7] for a comparison of the two languages when applied to the specification of the semantics of the small Pascal-like procedural language defined in [8]). In both cases, the definition of a programming language is derived by

- first constructing an abstract syntax,
- then specifying the consistency constraints (static semantics) that ensure that all language constructs (expressions, statements, etc.) are statically well formed (typically, well typed),
- and finally specifying the denotations (values) 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. If the programming language is enhanced, each of these structures must be extended, usually by extension of case analyses.

In this paper, we take a novel object-oriented approach to the specification of programming language semantics. Language constructs such as variables, expressions, statements, etc. are modelled as objects, with their own internal operations capturing the role of the construct within a program. A program, then, is specified in terms of its underlying constructs (i.e. the enclosed objects) and their interaction. As an illustration, the specification language Object-Z[5, 9] is used to define the semantics of a small procedural programming language similar to that defined in [8] but containing no block structuring.

Consequences of this object-oriented approach are

- one formal class structure captures the abstract syntax, static semantics and dynamic semantics of an individual language construct;
- the addition of a new construct typically only requires the definition of a new class.

In Section 2 the semantics of simple expressions is specified in both Z and Object-Z and the resulting formal structures compared. The Z specification makes use of the free-type construct[11] to capture abstractly the recursive nature of the syntax of expressions. The Object-Z specification, on the other hand, is based on a class-union construct[3] that facilitates the declaration of polymorphic classes. Those aspects of the construction needed for this paper are given in the Appendix.

Section 3 is an object-oriented specification of the syntax and semantics of the small procedural language. (Space limitations prevent the extension of the comparison with Z beyond that of Section 2). Notice that the language whose semantics is specified is not itself an object-oriented language.

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

bool	boolean	scal	scalar
const	constant	stmt	statement
decl	declaration	val	value
exp	expression	var	variable
int	integer	wf	well-formed

2 Semantics of Expressions in Z and Object-Z

A comparison of Z and Object-Z semantic definitions of expressions not involving variables is given in this section. (We assume the reader has a basic understanding of Z and Object-Z.)

2.1 Concrete Syntax of Expressions

The concrete syntax of expressions is presented in a BNF-like notation as follows:

$$
\begin{array}{rcl}
\mathit{Exp} & \rightarrow & \mathit{BoolConst} \\
 & & \mid \mathit{IntConst} \\
 & & \mid \mathit{Exp} \text{ `+' } \mathit{Exp} \\
 & & \mid \mathit{Exp} \text{ `<' } \mathit{Exp} \\
 & & \mid \mathit{Exp} \text{ `}\wedge\text{' } \mathit{Exp} \\
 & & \mid \text{`}\neg\text{ ' } \mathit{Exp} \\
\mathit{BoolConst} & \rightarrow & \text{`true'} \mid \text{`false'} \\
\mathit{IntConst} & \rightarrow & \mathit{Digit} \mid \mathit{Digit}\ \mathit{IntConst} \\
\mathit{Digit} & \rightarrow & \text{`0'} \mid \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \\
 & & \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'}
\end{array}
$$

Concrete syntax describes only external appearance. An abstract syntax and semantics are required to describe deeper structure and meaning.

2.2 Syntax and Semantics of Expressions in Z

The syntax of expressions is such that often

- an entity is defined to be one of several distinct entities;
- an entity is defined recursively.

In the following Z specification both these aspects are captured by the free-type construct[1, 10, 11]. An abstract syntax of expressions in Z is

$$
\begin{array}{rl}
\mathit{Exp} ::= & \mathit{boolconstexp}\langle\!\langle \mathbb{B} \rangle\!\rangle \\
 & \mid \mathit{intconstexp}\langle\!\langle \mathbb{Z} \rangle\!\rangle \\
 & \mid \mathit{plusexp}\langle\!\langle \mathit{Exp} \times \mathit{Exp} \rangle\!\rangle \\
 & \mid \mathit{lessexp}\langle\!\langle \mathit{Exp} \times \mathit{Exp} \rangle\!\rangle \\
 & \mid \mathit{andexp}\langle\!\langle \mathit{Exp} \times \mathit{Exp} \rangle\!\rangle \\
 & \mid \mathit{notexp}\langle\!\langle \mathit{Exp} \rangle\!\rangle
\end{array}
$$

However, not all expressions defined as above are well formed, e.g. $\mathit{true} + 12$ is allowed by the abstract syntax but is not well formed. Static semantic definitions are needed to capture well-formedness.

The scalar types of expression are boolean and integer:

$$\mathit{Scaltype} ::= \mathit{booltype} \mid \mathit{inttype}$$

The static semantics (type well-formedness) for expressions is defined as

$$
\begin{array}{|l}
\mathit{wf_sboolexp}, \mathit{wf_sintexp}, \mathit{wf_sexp} : \mathit{Exp} \rightarrow \mathbb{B} \\
\mathit{typeexp} : \mathit{Exp} \rightarrow\!\!\!\!\!+ \ \mathit{Scaltype} \\
\hline
\forall\, b : \mathbb{B};\ n : \mathbb{Z};\ e, e_1, e_2 : \mathit{Exp} \bullet \\
\quad \mathit{wf_sexp}(\mathit{boolconstexp}(b))
\end{array}
$$

$$
\begin{array}{l}
wf_s exp(intconstexp(n)) \\
wf_s exp(plusexp(e_1, e_2)) \\
\quad \Leftrightarrow wf_s intexp(e_1) \land wf_s intexp(e_2) \\
wf_s exp(lessexp(e_1, e_2)) \\
\quad \Leftrightarrow wf_s intexp(e_1) \land wf_s intexp(e_2) \\
wf_s exp(andexp(e_1, e_2)) \\
\quad \Leftrightarrow wf_s boolexp(e_1) \land wf_s boolexp(e_2) \\
wf_s exp(notexp(e)) \Leftrightarrow wf_s boolexp(e) \\
wf_s boolexp(e) \\
\quad \Leftrightarrow wf_s exp(e) \land typeexp(e) = booltype \\
wf_s intexp(e) \\
\quad \Leftrightarrow wf_s exp(e) \land typeexp(e) = inttype \\
\mathrm{dom}\, typeexp = \{e : Exp \mid wf_s exp(e)\} \\
typeexp(boolconstexp(b)) = booltype \\
typeexp(intconstexp(n)) = inttype \\
typeexp(plusexp(e_1, e_2)) = inttype \\
typeexp(lessexp(e_1, e_2)) = booltype \\
typeexp(andexp(e_1, e_2)) = booltype \\
typeexp(notexp(e)) = booltype
\end{array}
$$

The value of an expression can be either boolean or integer. However, in Z or Object-Z it is invalid to write

$$Value ::= \mathbb{B} \cup \mathbb{Z}.$$

Instead, the set of scalar (i.e. boolean or integer) values is defined using the free-type construct:

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

The meaning (value) of statically well-formed expressions is defined as:

$$
\begin{array}{l}
valexp : Exp \rightarrow\!\!\!\!\!\!+ \; Scalval \\
\hline
\mathrm{dom}\, valexp = \{e : Exp \mid wf_s exp(e)\} \\
valexp(boolconstexp(b)) = boolval(b) \\
valexp(intconstexp(n)) = intval(n) \\
valexp(plusexp(e_1, e_2)) = \\
\quad intval(intval^{\sim}(valexp(e_1)) + \\
\qquad intval^{\sim}(valexp(e_2))) \\
valexp(lessexp(e_1, e_2)) = \\
\quad boolval(intval^{\sim}(valexp(e_1)) < \\
\qquad intval^{\sim}(valexp(e_2))) \\
valexp(andexp(e_1, e_2)) = \\
\quad boolval(boolval^{\sim}(valexp(e_1)) \land \\
\qquad boolval^{\sim}(valexp(e_2))) \\
valexp(notexp(e)) = \\
\quad boolval(\neg\, boolval^{\sim}(valexp(e)))
\end{array}
$$

(The super-script $^{\sim}$ denotes function inverse.)

To add a new construct to *Exp*, e.g *orexp* (logical disjunction) or *multexp* (integer multiplication), we need to extend the abstract syntax, the static well-formedness and the value semantics. If we want to introduce variables into expressions, the specification needs to change dramatically because of the need to define a static environment, a dynamic environment and a store[2].

2.3 Syntax and Semantics of Expressions in Object-Z

We now present the syntax and semantics of *Exp* in Object-Z and show how the specification can be extended to accommodate logical disjunction and integer multiplication.

The idea is that each component of an expression can be modelled as an object. Constants are very obvious objects, but a binary (unary) expression can also be viewed as an object which contains two (one) sub-expression objects and has a value which can be derived from the values of the sub-expressions. An expression is either a constant, a plus-expression, a less-expression, an and-expression or a not-expression according to the concrete syntax of expressions in Section 2.1. In Object-Z this aspect is captured by the class-union construct (defined in the Appendix). In this case we have the class union

$$
\begin{array}{l}
Exp \mathrel{\widehat=} Const \cup PlusExp \cup LessExp \\
\qquad \cup AndExp \cup NotExp
\end{array}
$$

where *Const*, *PlusExp*, *LessExp*, *AndExp* and *NotExp* are classes specified below.

A function *type_of* between *Scalval* and *Scaltype* is defined by

$$
\begin{array}{l}
type\_of : Scalval \rightarrow Scaltype \\
\hline
\forall\, b : \mathbb{B};\; n : \mathbb{Z} \bullet \\
\quad type\_of(boolval(b)) = booltype \\
\quad type\_of(intval(n)) = inttype
\end{array}
$$

A constant is an object of the class *Const*:

[2]A static environment is a mapping from variable names to types; a dynamic environment is a mapping from variable names to locations: a store is a mapping from locations to scalar values.

Const

$$type : Scaltype$$
$$val : Scalval$$
$$type\_of(val) = type$$

The type of a constant is represented by attribute *type*, and the meaning (value) by attribute *val*. The class invariant ensures that the value and type of a constant are consistent.

Classes *PlusExp*, *LessExp* and *AndExp* of specific binary expression objects are defined by inheriting the generic binary expression *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 components.

BinaryExp

$$e_1, e_2 : Exp$$
$$type : Scaltype$$
$$e_1 \neq e_2$$
$$val : Scalval$$
$$type\_of(val) = type$$

UnaryExp

$$e : Exp$$
$$type : Scaltype$$
$$val : Scalval$$
$$type\_of(val) = type$$

The predicate $e_1 \neq e_2$ in the class *BinaryExp* indicates that the component sub-expressions are modelled as distinct objects, regardless of their individual syntax.

PlusExp

BinaryExp

$$type = inttype$$
$$e_1.type = inttype$$
$$e_2.type = inttype$$
$$val = intval(intval^{\sim}(e_1.val) + intval^{\sim}(e_2.val))$$

LessExp

BinaryExp

$$type = booltype$$
$$e_1.type = inttype$$
$$e_2.type = inttype$$
$$val = boolval(intval^{\sim}(e_1.val) < intval^{\sim}(e_2.val))$$

AndExp

BinaryExp

$$type = booltype$$
$$e_1.type = booltype$$
$$e_2.type = booltype$$
$$val = boolval(boolval^{\sim}(e_1.val) \wedge boolval^{\sim}(e_2.val))$$

NotExp

UnaryExp

$$type = booltype$$
$$e.type = booltype$$
$$val = boolval(\neg\ boolval^{\sim}(e.val))$$

In each expression class the attribute *val* depends in a functional way upon the other attributes. Such attributes are called *dependent* and are distinguished by being declared and defined in a separate 'open' schema immediately following the state schema[3].

In this specification, the abstract syntax, static semantics and dynamic semantics for each kind of expression are captured in the one class structure. In contrast, the Z specification distributes the same information over three different definitions.

The Ease of Reusability and Extendibility

A virtue of object orientation is that if we want to add, for example, binary constructions *OrExp* and *MultExp* it is only necessary to

[3]Dependent attributes never appear explicitly in the delta-list of operations, but are always included implicitly as their values depend upon the other attributes and hence are always subject to change.

(a) add the following classes:

$$\begin{array}{|l}
\hline
\textit{OrExp} \\
\textit{BinaryExp} \\
\hline
\textit{type} = \textit{booltype} \\
e_1.\textit{type} = \textit{booltype} \\
e_2.\textit{type} = \textit{booltype} \\
\hline
\textit{val} = \textit{boolval}(\textit{boolval}^{\sim}(e_1.\textit{val}) \vee \\
\qquad \textit{boolval}^{\sim}(e_2.\textit{val})) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{MultExp} \\
\textit{BinaryExp} \\
\hline
\textit{type} = \textit{inttype} \\
e_1.\textit{type} = \textit{inttype} \\
e_2.\textit{type} = \textit{inttype} \\
\hline
\textit{val} = \textit{intval}(\textit{intval}^{\sim}(e_1.\textit{val}) * \\
\qquad \textit{intval}^{\sim}(e_2.\textit{val})) \\
\hline
\end{array}$$

(b) add *OrExp* and *MultExp* into the class-union *Exp*. Other classes remain unchanged.

In Section 3 we shall show that variable references, too, can be easily included in expressions.

3 Language Semantics in Object-Z

3.1 Concrete Syntax

The concrete syntax of the small procedural language is:

$$\begin{array}{rcl}
\textit{Program} & \rightarrow & \text{`Program'}\ \textit{Id} \\
& & \text{`dec'}\ \textit{Declist}\ \textit{Stmt} \\
\textit{Id} & \rightarrow & \textit{Letter} \mid \textit{Letter}\ \textit{Id} \\
\textit{Letter} & \rightarrow & \text{`a'}\mid\text{`b'}\ldots \\
\textit{Declist} & \rightarrow & \epsilon \mid\ \textit{Declaration}\ \textit{Declist} \\
\textit{Declaration} & \rightarrow & \textit{Var}\ \text{`:'}\ \textit{Type}\ \text{`;'} \\
\textit{Var} & \rightarrow & \textit{Id} \\
\textit{Type} & \rightarrow & \text{`integer'} \mid \text{`boolean'} \\
\textit{Stmt} & \rightarrow & \textit{AssignStmt} \mid \textit{IfStmt} \mid \\
& & \textit{WhileStmt} \mid \textit{SeqStmt} \\
\textit{AssignStmt} & \rightarrow & \textit{Var}\ \text{`:='}\ \textit{Exp}\ \text{`;'} \\
\textit{IfStmt} & \rightarrow & \text{`if'}\ \textit{Exp}\ \text{`then'}\ \textit{Stmt} \\
& & \text{`else'}\ \textit{Stmt} \\
\textit{WhileStmt} & \rightarrow & \text{`while'}\ \textit{Exp}\ \text{`do'}\ \textit{Stmt} \\
\textit{Exp} & \rightarrow & \ldots\text{(as in Section 2.1)} \mid \textit{Var} \\
\textit{SeqStmt} & \rightarrow & \text{`begin'}\ \textit{Stmtseq}\ \text{`end'} \\
\textit{Stmtseq} & \rightarrow & \epsilon \mid \textit{Stmt}\ \textit{Stmtseq}
\end{array}$$

In the following subsections, the abstract syntax and static and dynamic semantics of this language are specified using Object-Z.

3.2 Expressions

The definition of expressions in Section 2.3 is extended by defining a variable reference class *Varref*. *Varref* requires the definition of the class *Var* which represents variables.

Before defining *Var* and *Varref* we need to add a dependent attribute *varrefs* to each component class of *Exp* (and later to each statement class) to gather applied occurrences of variable references. The totality of these references is used in the predicate part of the *Program* class (in Section 3.5) to ensure every variable appearing in a program statement also appears in a declaration.

$$\begin{array}{|ll}
\hline
\textit{Const} & \\
\ldots & \text{[as in Section 2.3]} \\
\textit{varrefs} : \mathbb{P}\ \textit{Varref}_{®} & \\
\hline
\textit{varrefs} = \varnothing & \\
\hline
\end{array}$$

$$\begin{array}{|ll}
\hline
\textit{BinaryExp} & \\
\ldots & \text{[as in Section 2.3]} \\
\textit{varrefs} : \mathbb{P}\ \textit{Varref}_{®} & \\
\hline
\textit{varrefs} = e_1.\textit{varrefs} \cup e_2.\textit{varrefs} & \\
\hline
\end{array}$$

$$\begin{array}{|ll}
\hline
\textit{UnaryExp} & \\
\ldots & \text{[as in Section 2.3]} \\
\textit{varrefs} : \mathbb{P}\ \textit{Varref}_{®} & \\
\hline
\textit{varrefs} = e.\textit{varrefs} & \\
\hline
\end{array}$$

The subscript $_{®}$ indicates that the object (or objects) identified are 'remote'. Other objects (declared without $_{®}$) are implicitly 'local', i.e. if a and b are distinct objects, regardless of their class, then the set of all objects local to a and those local to b are disjoint. Intuitively, if object a is local to object b then a is directly contained in b. However, no such containment notion applies to remote objects.

The classes *Var*, for modelling variables, and *Varref*, for modelling variable references, are now defined. *Varref* contains an identifier (*id*) and a reference (*var*) to a (remote) *Var* object. Let

$$[Id]$$

denote the set of all possible identifiers.

$$
\begin{array}{|l}
\textbf{Var} \\
\hline
\begin{array}{|l}
type : Scaltype \\
val : Scalval \\
\hline
type\_of(val) = type
\end{array} \\
\\
\begin{array}{|l}
\textit{INIT} \\
\hline
type = booltype \\
\quad \Rightarrow val = boolval(false) \\
type = inttype \\
\quad \Rightarrow val = intval(0)
\end{array} \\
\\
\begin{array}{|l}
Assign \\
\hline
\Delta(val) \\
val? : Scalval \\
\hline
val' = val?
\end{array} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textbf{Varref} \\
\hline
\begin{array}{|l}
id : Id \\
var : Var_{®}
\end{array} \\
\\
\begin{array}{|l}
type : Scaltype \\
val : Scalval \\
varrefs : \mathbb{P}\ Varref_{®} \\
\hline
type = var.type \\
val = var.val \\
varrefs = \{self\}
\end{array} \\
\\
Assign \mathrel{\widehat=} var.Assign \\
\hline
\end{array}
$$

When an object of class *Varref* is instantiated, the identifier *self* will denote the identity of the object itself (see [6] for a detailed discussion).

Note that variables differ from constants in that the attribute *val* can be changed (by the operation *Assign*). A boolean variable is initialized to *false* and an integer variable to 0.

The class-union *Exp* is extended by addition of *Varref* to give

$$
\begin{aligned}
Exp \mathrel{\widehat=} & \ Const \cup Varref \cup PlusExp \cup LessExp \\
& \cup AndExp \cup NotExp
\end{aligned}
$$

The definitions of *PlusExp*, *LessExp*, *AndExp* and *NotExp* remain as in Section 2.3.

3.3 Statements

A statement is an *assignment* statement, an *if* statement, a *while* statement or a *sequence* of statements:

$$
\begin{aligned}
Stmt \mathrel{\widehat=} & \ AssignStmt \cup IfStmt \cup \\
& WhileStmt \cup SeqStmt.
\end{aligned}
$$

The meaning of a statement, unlike an expression, is not a single value but rather a store transformation. In the semantics below, such transformations are effected by assigning values to variables.

An *assignment* statement is an object of the class *AssignStmt*:

$$
\begin{array}{|l}
\textbf{AssignStmt} \\
\hline
\begin{array}{|l}
varref : Varref \\
exp : Exp \\
\hline
varref.type = exp.type
\end{array} \\
\\
\begin{array}{|l}
varrefs : \mathbb{P}\ Varref_{®} \\
\hline
varrefs = \{varref\} \cup exp.varrefs
\end{array} \\
\\
\begin{array}{|l}
Val \\
\hline
val! : Scalval \\
\hline
val! = exp.val
\end{array} \\
\\
Execute \mathrel{\widehat=} Val \parallel varref.Assign \\
\hline
\end{array}
$$

The operation *Execute* evaluates *exp* and assigns it to the variable referenced by *varref*.

Note that the dependent attribute *varrefs* is the set of variable references appearing in an assignment statement. It serves a purpose similar to that of *varrefs* in expressions, namely, to ensure that every variable in an assignment statement is declared. It will be noted that other statements also have a *varrefs* dependent attribute.

An *if* statement is an object of the class *IfStmt*:

$$\textit{IfStmt}$$
$$test : Exp \quad then, else : Stmt \mid test.type = booltype$$
$$varrefs : \mathbb{P}\, Varref_{®} \mid varrefs = test.varrefs \cup then.varrefs \cup else.varrefs$$
$$Execute \mathrel{\widehat{=}} [boolval^{\sim}(test.val)] \land then.Execute \; [] \; [\neg\, boolval^{\sim}(test.val)] \land else.Execute$$

Operation *Execute* is defined using the symbol [] for operation choice[6]. The pre-conditions for each of the two alternatives of *Execute* are disjoint, so that either the *then* statement or the *else* statement is executed.

A *while* statement is an object of the class *WhileStmt*:

$$\textit{WhileStmt}$$
$$test : Exp \quad stmt : Stmt \mid test.type = booltype$$
$$varrefs : \mathbb{P}\, Varref_{®} \mid varrefs = test.varrefs \cup stmt.varrefs$$
$$Execute \mathrel{\widehat{=}} ([boolval^{\sim}(test.val)] \land stmt.Execute) \,\S\, Execute \; [] \; [\neg\, boolval^{\sim}(test.val)]$$

A *sequence* of statements is an object of the class *SeqStmt*:

$$\textit{SeqStmt}$$
$$stmts : \operatorname{seq} Stmt$$
$$varrefs : \mathbb{P}\, Varref_{®} \mid varrefs = \bigcup\{s : \operatorname{ran} stmts \bullet s.varrefs\}$$
$$Execute \mathrel{\widehat{=}} \S\, i : \operatorname{dom} stmts \bullet stmts(i).Execute$$

Note that the operation *Execute*, namely,

$$\S\, i : \operatorname{dom} stmts \bullet stmts(i).Execute,$$

is an abbreviation for

$$stmts(1).Execute \,\S\, stmts(2).Execute \,\S \ldots \S\, stmts(\#stmts).Execute.$$

3.4 The Program

The declaration part of a program is represented by an object of class *Vars* which contains a one-one mapping from identifiers to objects of class *Var*. This provides the required aggregation of variables (the store) and the unique association of each identifier with its corresponding variable.

$$\textit{Vars}$$
$$decs : Id \rightarrowtail Var \qquad \textit{INIT}: \forall v : \operatorname{ran} decs \bullet v.\textit{INIT}$$

A *program* is modelled as an object of the class *Program*. This completes the language semantics.

$$\textit{Program}$$
$$vars : Vars \quad stmt : Stmt \mid \{vr : stmt.varrefs \bullet vr.id \mapsto vr.var\} \subseteq vars.decs$$
$$\textit{INIT}: vars.\textit{INIT}$$
$$Execute \mathrel{\widehat{=}} stmt.Execute$$

As forshadowed, the class invariant ensures that applied occurrences of variable references have been declared.

Work has also been done by extending the context of the programming language to include procedures with both value and reference parameters.

4 Conclusion

In this paper we have taken an object-oriented approach to the specification of programming language semantics. The key idea introduced is to model language constructs such as expressions, variables, statements, etc. as objects. These objects belong to classes that have their own operations to capture the semantics of the corresponding language constructs. Using an object-oriented style to specify a programming language not only leads to a concise specification, but more importantly:

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

This paper is limited to an object-oriented specification of a simple procedural language. 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 *Modula*2, *Eiffel* and C^{++}.

Acknowledgements

We would like to thank Bernhard Baier, Susan Bray, Alena Griffiths, Wendy Johnston, Andreas Prinz and Margaret Turner for many helpful discussions on the issues raised in this paper.

References

[1] R. Arthan. On free type definitions in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, 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 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.

[4] R. Duke, D. Johnston, and G. Rose. Specifying the static semantics of block structured languages. *Australian Computer Journal*, 19(2):99–104, 1987.

[5] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice-Hall, 1991.

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

[7] I. Hayes. A small language definition in Z. Unpublished course notes, Computer Science Dept., University of Queensland, 1992.

[8] 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.

[9] 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.

[10] A. Smith. On recursive free types in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, University of York, December 1991.

[11] J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.

Appendix: Class Union

Object Identities

In the semantics of Object-Z each object has a unique persistent identity that distinguishes it from every other object, regardless of class[6, 9]. That is, an object's identity is invariant irrespective of changes to its state.

If A is the name of a class, let A also denote the set of identities of possible objects of class A[4], i.e. the declaration $a : A$ means that a identifies an object of class A. In other words, the *type* of class A is the set of all identities of objects

[4] 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 iden tities of objects of that class.

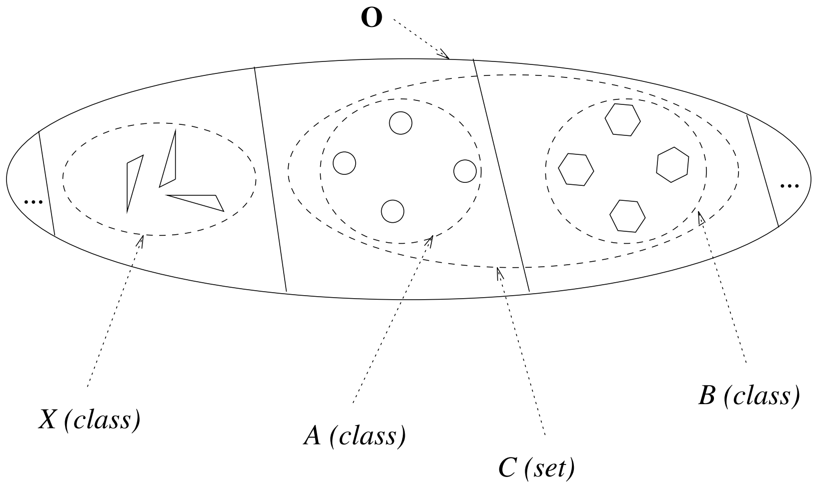

Figure 1: Class Union

in that class. Let $\mathbb{O}$ denote the set of all object identities, regardless of class, that may be in a system.[5]

If A and B are distinct classes then

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

i.e. the objects of class A and the objects of class B are distinct 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.

Class Union

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
$\{obj : \mathbb{O} \mid obj \in A \vee obj \in B\}$ of objects in $\mathbb{O}$.

If we define $C \mathrel{\widehat{=}} A \cup B$ then C is not itself a class: it is a set of object identities as shown in Figure 1. C is called a *class union*.

Class unions such as C can also be components in other class union definitions.

[5] In any system only a finite number of distinct classes will be specified.

Polymorphic Features

Associated with any object is a set of operations and attributes (features).

Given a collection of objects, possibly from different classes, an operation named *Op*, say, is said to be *polymorphic with respect to the collection* if

- every object in the collection has an associated operation named *Op*;
- for each object in the collection, the interface of the operation *Op* is the same, i.e. the input and output parameters are identically named and typed.

Similarly, an attribute is polymorphic with respect to a collection if each object in the collection has an attribute of that name and type.

The set of all features that are polymorphic with respect to a collection of objects is called its *polymorphic core*.

Polymorphism and Class Union

In contrast to polymorphism induced by inheritance, class unions can be defined, and polymorphism induced, on arbitrary classes. For example, given the class union C as defined above, a declaration $c : C$ indicates that c is a polymorphic object identity in that the value of c is a identity of an object in either the class A or B. Furthermore, an expression of the form $c.x$

is only possible if x is an attribute or operation in the polymorphic core of C.