

SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-48

A Logic for Object-Z

Graeme Smith

December 1994

Phone: +61 7 365 1003

Fax: +61 7 365 1533

A Logic for Object-Z

Graeme Smith

Software Verification Research Centre

Department of Computer Science

University of Queensland

Australia

Abstract

This paper presents a logic for Object-Z which extends \mathcal{W} , the logic for Z adopted as the basis of the deductive system in the Z Base Standard. The logic provides a basis on which tool support for reasoning about Object-Z specifications can be developed. It also formalises the intended meaning of Object-Z constructs and hence provides an abstract, axiomatic semantics of the language.

1 Introduction

Object-Z[10, 11, 7] is an extension of Z in which the existing syntax and semantics of Z are retained and new constructs are introduced to facilitate specification in an object-oriented style. The enhanced structuring improves the readability of large specifications. It also enables the possibility of modular verification and refinement. This is dependent, however, on the development of a formal semantics, a proof system and, ultimately, tools for Object-Z.

Since Object-Z is a conservative extension of Z, in the sense that the existing syntax and semantics of Z are retained, a natural approach to developing a semantics has been to extend already existing semantics of Z. A denotational semantics of an early version of Object-Z[6] which extended the denotational semantics of Z of Spivey[12] was described in [5]. More recently, a denotational semantics of Object-Z extending that of the Z Base Standard[1] has been proposed[8].

Similarly, an axiomatic semantics or logic for reasoning about Object-Z specifications could be developed as an extension to an existing logic for Z. Approaches for reasoning in Z have been proposed by Spivey[12], Woodcock and Loomes[16], Diller[2] and Wordsworth[17] but none of these attempt to give a complete set of proof rules for Z. Progress towards such a set of rules, however, has been made in \mathcal{W} [15]. These rules have been adopted by the Z Base Standard and (largely) proved sound with respect to its denotational semantics. This proof of soundness together with the fact that \mathcal{W} has been successfully encoded on several theorem provers (e.g. see [9]) makes it an ideal choice as the basis for a logic for Object-Z.

The objective of this paper is to extend the rules in \mathcal{W} to cover the additional constructs in Object-Z. This is facilitated by extensions to \mathcal{W} itself. Section 2 introduces Object-Z and Section 3 provides an overview of \mathcal{W} and details the extensions adopted to accommodate Object-Z. Sections 4 to 7 present the proof rules for reasoning about Object-Z specifications.

Throughout the paper, the following naming conventions are adopted.

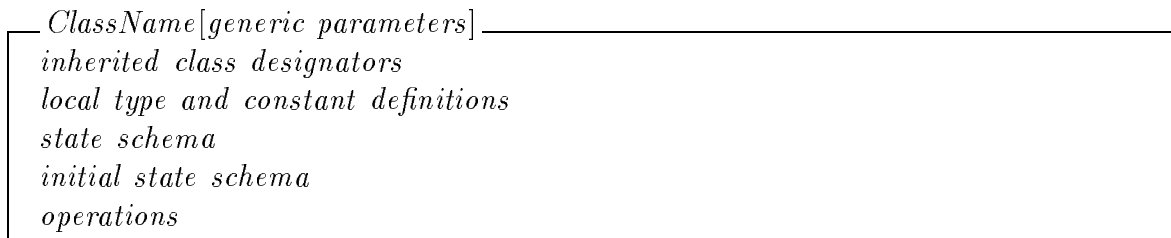
d	declarations
p, q	predicates
Φ, Ψ	sets of predicates
s, t, u, v	expressions
x, y, z	variables
a	object references
b	bindings
S	schemas
A, B	classes
OP	operations
Op	operation names
T	types
X	generic types
D	type or axiomatic definitions
λ	rename lists

2 Object-Z

The major extension in Object-Z is the *class schema* which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which may affect its variables. The class schema is not simply a syntactic extension but also defines a type whose instances are *object references*, i.e. identifiers which reference objects of the class¹. By declaring variables of class types within a class, objects which refer to and use other objects may be specified.

An Object-Z class schema, often referred to simply as a class, is represented syntactically as a named box possibly with generic parameters. In this box there may be local type and constant definitions, at most one state schema and associated initial state schema, and zero or more operations. In addition to these explicit definitions, a class may *inherit* the definition of one or more other classes. It may also have a visibility list restricting the way objects of the class may be used, and a history invariant for capturing liveness properties; these, however, are not discussed in this paper.

For this paper, the basic structure of a class is as follows.



¹In an early version of Object-Z[6], instances of a class schema were objects of the class. However, work on the language, aimed at implicit support for object identity and object sharing, has motivated a revised semantics based on object references.

A class designator consists of a class name, an instantiation of that class' generic parameters and possibly a rename list. The rename list allows constants, state variables and operations to be renamed. For example, given a generic class $A[X_1, X_2]$ with state variables x and y and operations Op_1 and Op_2 , the following is a class designator which instantiates the generic types X_1 and X_2 with actual types t_1 and t_2 and renames y to z and Op_2 to Op_3 .

$$A[t_1, t_2][z/y, Op_3/Op_2]$$

When a class is inherited, its local types and constants are implicitly available in the inheriting class. Any types or constants with the same name occurring in both the inherited and the inheriting class are semantically identified and hence must have compatible definitions.

The inherited class' state schema and initial state schema are implicitly conjoined with those declared explicitly in the inheriting class. The inherited class' operations are implicitly available except where the name of an operation in the inheriting class is the same as that of an operation of the inherited class. In this case, the inherited operation is implicitly conjoined with that in the inheriting class.

The local type and constant definitions of a class have the same syntax as global type and constant definitions in Z . Their scope, however, is limited to the class in which they are declared. A constant is associated with a fixed value which cannot be changed by any operations of the class. However, the value of constants may differ for different objects of the class.

The state schema is nameless and has its declarations partitioned by a Δ into *primary* and *secondary* variables. Secondary variables are implicitly available for change in any operation and are usually defined in terms of the primary variables. For example, the state schema of a 'square' class may have *side* as a primary variable and *area* and *perimeter* as secondary variables.

$side : \mathbb{N}$ Δ $area : \mathbb{N}$ $perimeter : \mathbb{N}$
$area = side * side$ $perimeter = 4 * side$

Both constants and state variables can be object references. They are declared using a class designator either as a reference to a member of a particular class, e.g. $a : A[t_1, t_2]$, or to a member of a collection of classes belonging to a particular inheritance hierarchy², e.g. $a : \downarrow A[t_1, t_2]$ is a reference to an object of class $A[t_1, t_2]$ or any class derived from $A[t_1, t_2]$ by inheritance. The constants and variables of an object can be accessed using the dot notation, e.g. $a.x$ denotes the constant or variable x of the object referenced by a . The dot notation is also used to specify that an object is in its initial state, e.g. $a.INIT$, and to apply an operation to an object, e.g. $a.Op_1$.

The initial state schema is named with the keyword *INIT* and has only a predicate part. It implicitly includes both the declarations and the predicates of the state schema.

²Object references to a member of a class in an arbitrary collection of classes[3] or to objects "contained" in a class[4] can also be declared. These notions will not be discussed in this paper.

The operations are defined either as *operation schemas* or *operation expressions*. They are interpreted in the class' local environment (i.e. the global environment of the specification enriched with the local type and constant definitions) enriched with the declarations and predicates of the state schema in both primed and unprimed form. Therefore, if $a : A[t_1, t_2]$ is declared in the state schema of a class, $a.Op_1$ is a valid operation definition.

An operation schema extends the notion of a schema in Z by adding to it a *delta-list*. The delta-list is a list of the primary variables which the operation may change when it is applied to an object of the class; all other primary variables remain unchanged³. When two or more operations are combined to define a new operation, however, their delta-lists are united so that the new operation may change any variable which any of its constituent operations could have changed. In this way, delta-lists enable a more flexible calculus for combining operations than would be allowed by Z schemas alone.

For example, given the operations Inc_x and Inc_y which increment the variables x and y respectively,

$$\frac{Inc_x \quad \Delta(x)}{x' = x + 1} \qquad \frac{Inc_y \quad \Delta(y)}{y' = y + 1}$$

the operation expression $Inc_x \wedge Inc_y$ is equivalent to the following operation schema.

$$\frac{Inc_x \wedge Inc_y \quad \Delta(x, y)}{x' = x + 1 \quad y' = y + 1}$$

Apart from conjunction \wedge , other operation operators are the *parallel operator* \parallel , the *choice operator* \square , the *enrichment operator* \bullet , hiding and renaming⁴.

The parallel operator \parallel is a binary operator introduced into Object-Z to allow specification of inter-object communication. The operator identifies and equates input variables in either operation with output variables in the other operation having the same basename, i.e. apart from the $?$ or $!$. The identified input variables are hidden in the resulting operation; the output variables are not hidden and so may be equated with other input variables in subsequent parallel compositions.

The choice operator \square is a binary operator which allows the specification of nondeterministic choice between operations. The meaning of $Op_1 \square Op_2$ is that either operation Op_1 occurs or operation Op_2 occurs but not both. For example, $Inc_x \square Inc_y$ is equivalent to the following operation schema.

$$\frac{Inc_x \square Inc_y \quad \Delta(x, y)}{(x' = x + 1 \wedge y' = y) \vee (y' = y + 1 \wedge x' = x)}$$

³When no delta-list appears in an operation, this is equivalent to having an empty delta-list, i.e. no primary variables can change.

⁴Object-Z also has a sequential composition operator which will not be discussed in this paper.

The enrichment operator \bullet allows operations to be interpreted within the class' local environment enriched with the declarations and predicates of another operation or schema. It is particularly useful when specifying operations occurring on particular objects in a set of objects: references to the selected objects appear in the declarations of the first operation and can be used in the definition of the second. For example, given the declaration $s : \mathbb{P} A[t_1, t_2]$ in the state schema of a class, the operation which involves the parallel composition of two distinct objects referenced from s performing Op_1 is as follows.

$$[a_1, a_2 : s \mid a_1 \neq a_2] \bullet a_1.Op_1 \parallel a_2.Op_1$$

Hiding and renaming are defined as in Z . Since the primary and secondary variables (in both primed and unprimed form) are only available to an operation definition (and not part of it) they cannot be hidden or renamed.

3 \mathcal{W}

\mathcal{W} is a Gentzen-style sequent calculus. Axioms and theorems are expressed using *sequents* and further theorems are derived by the application of *inference rules*. The expression of these rules is facilitated by the use of *meta-functions* defined outside the logic.

3.1 Sequents

Sequents are of the form

$$d \mid \Psi \vdash \Phi$$

The *antecedent* of the sequent consists of a list of declarations d and a set of predicates Ψ and the *consequent* is a set of predicates Φ . A sequent is *valid* if at least one of the predicates in Ψ is true in the global environment of the specification enriched by d and satisfying all predicates in Φ . Both the antecedent and the consequent can be empty. An empty antecedent is equivalent to the empty declaration and the single predicate *true*. An empty consequent is equivalent to the single predicate *false*. (In effect, predicates in Ψ are conjoined whereas predicates in Φ are disjoined.)

For example, given a Z specification which includes the schema

$$\frac{S}{\frac{x, y : \mathbb{N}}{x < y}}$$

the following sequents are valid. (S is a declaration in these sequents, i.e. it abbreviates $x, y : \mathbb{N} \mid x < y$ as in Z .)

$$\begin{aligned} S \vdash x \in \mathbb{N} \\ S \vdash y \in \mathbb{N} \\ S \vdash x < y \end{aligned}$$

An Object- Z specification comprises not only a global environment but also a local environment for each specified class. To reason about properties within these local environments,

we extend \mathcal{W} so that sequents can be interpreted within a particular class context as follows.

$$A :: d \mid \Psi \vdash \Phi$$

The validity of the sequent is determined in the global environment of the specification enriched with the local environment of class A . For example, if the specification contains the following class

$$\boxed{\begin{array}{l} A \\ \hline \begin{array}{l} x : \mathbb{N} \\ \hline x < 10 \\ \vdots \end{array} \end{array}}$$

then the following are valid sequents.

$$\begin{array}{l} A :: \vdash x \in \mathbb{N} \\ A :: \vdash x < 10 \end{array}$$

3.2 Inference rules

Inference rules for manipulating sequents are of the form

$$\frac{\text{premisses}}{\text{conclusion}} \quad (\text{name}) [\text{proviso}]$$

The *premisses* are zero or more sequents and the *conclusion* is a single sequent. The *name* identifies the rule and the *proviso* is a predicate which must be true in the global environment of the specification for the rule to be applicable. An inference rule is *sound* if, whenever the proviso is satisfied and the premisses are valid, the conclusion is also valid.

As an example, consider the following rule for logical disjunction.

$$\frac{p_1 \vdash \quad p_2 \vdash}{p_1 \vee p_2 \vdash} \quad (\vee \vdash)$$

Such a rule (with empty consequents) is intended to be used in combination with \mathcal{W} 's "rule-lifting" meta-theorem. This meta-theorem allows a common declaration to be added to every antecedent in a rule or a common predicate to be added to either every antecedent or every consequent. Therefore, given that the above rule is sound, so is the following rule which is derived by adding the predicate q to each consequent.

$$\frac{p_1 \vdash q \quad p_2 \vdash q}{p_1 \vee p_2 \vdash q}$$

This rule states that if we can deduce the predicate q from the predicates p_1 and p_2 then we can deduce q from the predicate $p_1 \vee p_2$.

We extend the notation for representing provisos so that they can be interpreted in a class context. For example, the proviso $A :: p$ states that p must be true in the global environment of the specification enriched with the local environment of the class A .

So that \mathcal{W} 's inference rules can be used in the context of a class, the following meta-theorem is introduced.

If the rule
$$\frac{d_1 \mid \Psi_1 \vdash \Phi_1}{d_2 \mid \Psi_2 \vdash \Phi_2} [p]$$
 is sound,

then the rule
$$\frac{A :: d_1 \mid \Psi_1 \vdash \Phi_1}{A :: d_2 \mid \Psi_2 \vdash \Phi_2} [A :: p]$$
 is also sound.

This meta-theorem also allows us to define rules involving a single class without explicitly stating a class context.

For each existing meta-theorem of \mathcal{W} (e.g. “rule-lifting”), we also introduce a similar meta-theorem to hold within a given class context.

3.3 Meta-functions and operators

\mathcal{W} makes use of meta-functions to facilitate the expression of rules. The meta-function α returns the names of the variables declared in a declaration or schema expression. The meta-function ϕ returns the names of free variables in an expression, predicate, declaration or schema expression.

\mathcal{W} also has a meta-substitution operator \odot ⁵ which allows a binding to be substituted into an expression, predicate or declaration. When a binding is substituted into an expression, the free variables in the expression which are in the domain of the binding are replaced by the corresponding expressions in the range of the binding. For example, if the domain of the binding b contains x and y then:

$b \odot (x \in y)$ evaluates to $b.x \in b.y$

When substituting into a declaration the substitution is made into the types only, i.e. for b as above:

$b \odot (x : y)$ evaluates to $x : b.y$

The meta-substitution operator is used to define renaming in \mathcal{W} . For example, to rename x to y in the predicate p the following notation is adopted.

$\langle x \rightsquigarrow y \rangle \odot p$

To simplify notation, it is assumed that a rename list is of the form $x_1 \rightsquigarrow s_1, \dots, x_n \rightsquigarrow s_n$. Thus, the predicate of $S[\lambda]$ where S is defined as in Section 3.1 is denoted

$\langle \lambda \rangle \odot (x < y)$

⁵We adopt the symbol ‘ \odot ’ used in the Z Base Standard rather than the symbol ‘.’ used in \mathcal{W} .

The meta-function α is extended for Object-Z to also return the names of the types⁶ and attributes (i.e. constants and primary and secondary variables) of a class. This set includes both the types and attributes explicitly declared in the class and those implicitly included via inherited classes. To define it, we therefore need first to define a meta-function ι which returns the set of inherited classes of a class. This meta-function needs to account for the fact that the actual parameters of an inherited class may depend on the generic parameters of the class into which it is inherited. For example, if the class $B[X_1]$ inherits the class $A[X_2]$ with its generic parameter X_2 instantiated with X_1 then the set of classes inherited by the class $B[t]$ will include $A[t]$.

In general, given a class $B[X_1, \dots, X_n]$ with inherited classes $A_1[u_1, \dots, u_j], \dots, A_m[v_1, \dots, v_k]$, we have

$$\iota(B[t_1, \dots, t_n]) = \{A_1[b \odot u_1, \dots, b \odot u_j], \dots, A_m[b \odot v_1, \dots, b \odot v_k]\} \\ \text{where } b = \langle X_1 \rightsquigarrow t_1, \dots, X_n \rightsquigarrow t_n \rangle$$

α is then defined for classes as follows where α_d is an additional meta-function which returns the names of the explicitly declared (i.e. non-inherited) types and attributes of a class.

$$\alpha(A[\lambda]) = \langle \lambda \rangle \odot \alpha(A) \\ \alpha(B) = \alpha_d(B) \cup \alpha(A_1) \cup \dots \cup \alpha(A_n) \quad \text{where } \iota(B) = \{A_1, \dots, A_n\}$$

In a similar fashion, the meta-functions α_1 and α_2 are defined to return the names of the primary and secondary variables of a class respectively.

We also extend the meta-function ϕ and the meta-substitution operator \odot of \mathcal{W} to treat references to the state schema, initial state schema and operations of a class as variables rather than schema references. This enables us to rename class operations and is also necessary to define inference rules where we want to use definitions from one class in the context of another. In such cases, the definitions must not include (as free variables) references to the class' schemas as these will, in general, have a different meaning in the other class context. Examples of these uses of ϕ and \odot appear in the following sections.

Further meta-functions and operators required for the rules in this paper will be defined as they are needed.

4 Classes

Given the definition of a generic class $A[X_1, \dots, X_n]$, we have a set of inference rules from which properties of the class can be derived. These rules define the class' local environment, state schema, initial state schema and operations.

4.1 Local definitions

The meta-function η is introduced to return a predicate representing the properties of the local definitions of a class. These properties are inherited with a class, whereas other

⁶In the case of a free type, the names of the constants in its axiomatic representation (see Spivey[13] for details) are also returned by α .

properties (e.g. those derived by induction) are not. η is defined in terms of a meta-function η_d which returns the properties of the explicitly declared local definitions. It is defined, in turn, by a meta-function η_D which returns the properties of individual type or axiomatic definitions as follows (\equiv is meta-equivalence).

$$\begin{aligned}
\eta_D[T_1, \dots, T_n] &\equiv \text{true} \\
\eta_D(T == s) &\equiv T = s \\
\eta_D(d \mid p) &\equiv [d] \wedge p \\
\eta_D(T ::= x_1 \mid \dots \mid x_m \mid y_1 \langle\langle s_1[T] \rangle\rangle \mid \dots \mid y_k \langle\langle s_k[T] \rangle\rangle) &\equiv \\
&x_1 \in T, \dots, x_m \in T \\
&y_1 \in s_1[T] \mapsto T \wedge \dots \wedge y_k \in s_k[T] \mapsto T \\
&\text{disjoint}(\{x_1\}, \dots, \{x_m\}, \text{ran } y_1, \dots, \text{ran } y_k) \\
&\forall z : \mathbb{P} T \bullet \{x_1, \dots, x_m\} \cup y_1(\mid s_1[z] \mid) \cup \dots \cup y_k(\mid s_k[z] \mid) \subseteq z \Rightarrow T \subseteq z
\end{aligned}$$

In defining the properties of an axiomatic definition we place schema brackets around its declaration to form a schema which can then be used as a predicate. The properties of a free type are derived from its axiomatic definition as given in Spivey[13].

Given that $A[X_1, \dots, X_n]$ has local definitions D_1, \dots, D_n ,

$$\begin{aligned}
\eta_d(A[t_1, \dots, t_n]) &\equiv b \odot (\eta_D(D_1) \wedge \dots \wedge \eta_D(D_n)) \\
\text{where } b &= \langle\langle X_1 \rightsquigarrow t_1, \dots, X_n \rightsquigarrow t_n \rangle\rangle
\end{aligned}$$

To account for inherited classes (possibly with renaming) we have

$$\begin{aligned}
\eta(A[\lambda]) &\equiv \langle\langle \lambda \rangle\rangle \odot \eta(A) \\
\eta(B) &\equiv \eta_d(B) \wedge \eta(A_1) \wedge \dots \wedge \eta(A_n) \quad \text{where } \iota(B) = \{A_1, \dots, A_n\}
\end{aligned}$$

Whenever we are reasoning within the local environment of class $A[X_1, \dots, X_n]$ instantiated with actual parameters t_1, \dots, t_n , we can use the axioms describing the local types and axiomatic definitions. This is captured by the following rule.

$$\frac{A[t_1, \dots, t_n] :: \quad \eta(A[t_1, \dots, t_n]) \vdash}{A[t_1, \dots, t_n] :: \quad \vdash} \quad (\text{LocalDefs})$$

For example, assuming class A of Section 3.1 has no other local definitions,

$$\begin{aligned}
\eta(A) &\equiv [x : \mathbb{N}] \wedge x < 10 \\
&\equiv x \in \mathbb{N} \wedge x < 10
\end{aligned}$$

To prove the sequent $A :: \vdash x \in \mathbb{N}$ is valid, we use the *LocalDefs* rule in combination with the “rule-lifting” meta-theorem to obtain the following rule.

$$\frac{A :: \quad x \in \mathbb{N} \wedge x < 10 \vdash x \in \mathbb{N}}{A :: \quad \vdash x \in \mathbb{N}}$$

Since the premiss is obviously valid, the conclusion must also be valid.

4.2 State schema

The keyword *STATE* is introduced to identify the schema corresponding to the (mutable) state of the class. This schema consists of the inherited state schema extended with the primary and secondary variable declarations and predicate of the explicit state schema.

To refer to the inherited state schema the name $\uparrow STATE$ is used. Similarly, the name $\uparrow INIT$ refers to the inherited part of the initial state schema and $\uparrow Op$ to the inherited part of an operation *Op*.

If $A[X_1, \dots, X_n]$ has the state schema

d_1
Δ
d_2
p

we have

$$\frac{A[t_1, \dots, t_n] :: \quad STATE = [\uparrow STATE; b \odot d_1; b \odot d_2 \mid b \odot p] \vdash}{A[t_1, \dots, t_n] :: \quad \vdash} \quad (StateDef) [q]$$

$$\text{where } q \equiv b = \langle \langle X_1 \rightsquigarrow t_1, \dots, X_n \rightsquigarrow t_n \rangle \rangle$$

The antecedent of the premiss is a predicate equating the two schema expressions. This rule is intended to be used in combination with “rule-lifting” in a similar fashion to *LocalDefs*.

4.3 Initial state schema

The schema corresponding to the initial state of a class consists of the inherited initial state schema conjoined with the state schema of the class extended with the predicate of the explicit initial state schema.

If $A[X_1, \dots, X_n]$ has the initial state schema

$INIT$
p

we have

$$\frac{A[t_1, \dots, t_n] :: \quad INIT = \uparrow INIT \wedge [STATE \mid b \odot p] \vdash}{A[t_1, \dots, t_n] :: \quad \vdash} \quad (InitDef) [q]$$

$$\text{where } q \equiv b = \langle \langle X_1 \rightsquigarrow t_1, \dots, X_n \rightsquigarrow t_n \rangle \rangle$$

4.4 Operations

For notational convenience, whenever an operation appears as an expression, predicate, declaration or schema expression in the logic, we interpret it as the schema part of the operation only (i.e. without the delta-list).

The schema part of an operation consists of the inherited part of the operation conjoined with the schema part of the explicit operation definition interpreted in the local environment of the class enriched with $\Delta STATE$ (i.e. $STATE \wedge STATE'$).

For each operation definition

$$Op \cong OP$$

in $A[X_1, \dots, X_n]$ we have

$$\frac{A[t_1, \dots, t_n] :: \quad Op = \Delta STATE \bullet (\uparrow Op \wedge b \odot OP) \vdash}{A[t_1, \dots, t_n] :: \quad \vdash} \quad (OpDef) [p]$$

$$\text{where } p \equiv b = \langle X_1 \rightsquigarrow t_1, \dots, X_n \rightsquigarrow t_n \rangle$$

The meta-function Ω is introduced to return the set of names of operations of a class. It is defined in terms of a function Ω_d which returns the explicitly defined operations of the class.

$$\begin{aligned} \Omega(A[\lambda]) &= \langle \lambda \rangle \odot \Omega(A) \\ \Omega(B) &= \Omega_d(B) \cup \Omega(A_1) \cup \dots \cup \Omega(A_n) \quad \text{where } \iota(B) = \{A_1, \dots, A_n\} \end{aligned}$$

If a particular operation name Op occurs in one or more inherited classes but not in the inheriting class then there is an implicit operation named Op in the inheriting class. The schema part of Op consists of the schema part of the inherited operation interpreted in the local environment of the class enriched with $\Delta STATE$.

$$\frac{A[t_1, \dots, t_n] :: \quad Op = \Delta STATE \bullet \uparrow Op \vdash}{A[t_1, \dots, t_n] :: \quad \vdash} \quad (ImplicitOpDef) [p]$$

$$\begin{aligned} \text{where } p &\equiv \iota(A[t_1, \dots, t_n]) = \{A_1, \dots, A_m\} \\ &\quad Op \in \Omega(A_1) \cup \dots \cup \Omega(A_m) \\ &\quad Op \notin \Omega_d(A) \end{aligned}$$

5 Inheritance

The inherited part of a schema is defined as the conjunction of the same-named schemas of each inherited class. Before the conjunction takes place, each schema must be fully expanded so that it does not include references to other schemas of its class.

5.1 Inherited state schema

The inherited state schema of a class is the conjunction of each of the state schemas of the inherited classes.

$$\frac{A_1 :: \vdash STATE = S_1 \quad \dots \quad A_n :: \vdash STATE = S_n}{B :: \vdash \uparrow STATE = S_1 \wedge \dots \wedge S_n} \quad (\uparrow STATE) [p]$$

$$\begin{aligned} \text{where } p &\equiv \iota(B) = \{A_1, \dots, A_n\} \\ &\phi(S_1) \subseteq \alpha(A_1) \\ &\dots \\ &\phi(S_n) \subseteq \alpha(A_n) \end{aligned}$$

The provisos of the form $\phi(S_i) \subseteq \alpha(A_i)$ ensure that the schemas S_i are expressed in fully expanded form (i.e. in terms of $\alpha(A_i)$ only and not in terms of references to the state, initial state or operations of A_i).

5.2 Inherited initial state schema

The inherited initial state schema of a class is the conjunction of each of the initial state schemas of the inherited classes.

$$\frac{A_1 :: \vdash INIT = S_1 \quad \dots \quad A_n :: \vdash INIT = S_n}{B :: \vdash \uparrow INIT = S_1 \wedge \dots \wedge S_n} \quad (\uparrow INIT) [p]$$

$$\begin{aligned} \text{where } p &\equiv \iota(B) = \{A_1, \dots, A_n\} \\ &\phi(S_1) \subseteq \alpha(A_1) \\ &\dots \\ &\phi(S_n) \subseteq \alpha(A_n) \end{aligned}$$

5.3 Inherited operations

The schema of the inherited part of an operation is the conjunction of each of the schemas of the operations of the same name occurring in the inherited classes. Not all of the inherited classes need have such an operation.

$$\frac{A_1 :: \vdash Op = S_1 \quad \dots \quad A_k :: \vdash Op = S_k}{B :: \vdash \uparrow Op = S_1 \wedge \dots \wedge S_k} \quad (\uparrow Op) [p]$$

$$\begin{aligned} \text{where } p &\equiv \iota(B) = \{A_1, \dots, A_n\} \\ &Op \in \Omega(A_1) \cap \dots \cap \Omega(A_k) \\ &Op \notin \Omega(A_{k+1}) \cup \dots \cup \Omega(A_n) \\ &\phi(S_1) \subseteq \alpha(A_1) \\ &\dots \\ &\phi(S_k) \subseteq \alpha(A_k) \end{aligned}$$

5.4 Renaming

An inherited class may have attributes and operations renamed. The properties of the resulting class are obtained by applying the renaming to the properties of the original class.

$$\frac{A :: \vdash p}{A[\lambda] :: \vdash \langle \lambda \rangle \odot p} \quad (\textit{ClassRenaming})$$

6 Operation Expressions

The properties of the schema parts of operation expressions can be expressed in terms of Z schema expressions. These expressions can then be reasoned about using the rules of \mathcal{W} .

To avoid ambiguity, the schema part of an operation is denoted explicitly by the notation $[OP]$ (i.e. the schema which includes (the schema of) OP as a declaration) where necessary. For example, $[OP_1] \wedge [OP_2]$ is a schema expression whereas $OP_1 \wedge OP_2$ is an operation expression.

6.1 Conjunction

The schema of the operation $OP_1 \wedge OP_2$ is the conjunction of the schemas of OP_1 and OP_2 .

$$\frac{OP_1 \wedge OP_2 = [OP_1] \wedge [OP_2] \vdash}{\vdash} \quad (\textit{OpConjunction})$$

6.2 Parallel

To define the parallel operator, the meta-functions $\beta_?$ and $\beta_!$ are introduced to return the basenames of the inputs and outputs of an operation respectively.

The schema of the operation $OP_1 \parallel OP_2$ can be formed by

- noting any input variables of the schema of each operation which have the same basename as an output variable of the schema of the other operation,
- renaming each such input variable to the name of the output variable,
- and conjoining the resulting schemas.

$$\frac{OP_1 \parallel OP_2 = [OP_1][y_1!/y_1?, \dots, y_m!/y_m?] \wedge [OP_2][x_1!/x_1?, \dots, x_n!/x_n?] \vdash}{\vdash} \quad (\textit{Parallel}) [p]$$

$$\text{where } p \equiv \beta_!(OP_1) \cap \beta_?(OP_2) = \{x_1, \dots, x_n\}$$

$$\beta_?(OP_1) \cap \beta_!(OP_2) = \{y_1, \dots, y_m\}$$

6.3 Choice

To define the choice operator, the meta-function δ_{OP} is introduced to return the names of the variables in the delta-list of an operation.

$$\begin{aligned}
\delta_{OP} [\Delta(x_1, \dots, x_n) \ d \mid p] &= \{x_1, \dots, x_n\} \\
\delta_{OP} [d \mid p] &= \emptyset \\
\delta_{OP}(a.Op) &= \emptyset \\
\delta_{OP}(OP_1 \wedge OP_2) &= \delta_{OP}(OP_1) \cup \delta_{OP}(OP_2) \\
\delta_{OP}(OP_1 \parallel OP_2) &= \delta_{OP}(OP_1) \cup \delta_{OP}(OP_2) \\
\delta_{OP}(OP_1 \parallel\!\! \parallel OP_2) &= \delta_{OP}(OP_1) \cup \delta_{OP}(OP_2) \\
\delta_{OP}(OP_1 \bullet OP_2) &= \delta_{OP}(OP_1) \cup \delta_{OP}(OP_2) \\
\delta_{OP}(OP[\lambda]) &= \delta_{OP}(OP) \\
\delta_{OP}(OP \setminus (x_1, \dots, x_n)) &= \delta_{OP}(OP)
\end{aligned}$$

Note that the delta-list of $a.Op$ is empty because this operation doesn't change any of the variables of the class in which it occurs. Although the object referenced by a is changed, the reference a itself is not. Also, the delta-list of an operation is not changed by renaming or hiding since these do not affect the class' variables.

The schema of the operation $OP_1 \parallel\!\! \parallel OP_2$ can be formed by

- adding to the schema of OP_1 the predicate which states that all variables in OP_2 's delta-list which are not in OP_1 's delta-list are unchanged,
- adding to the schema of OP_2 the predicate which states that all variables in OP_1 's delta-list which are not in OP_2 's delta-list are unchanged,
- and disjoining the resulting schemas.

$$\frac{
\begin{array}{c}
OP_1 \parallel\!\! \parallel OP_2 = [OP_1 \mid x'_1 = x_1 \wedge \dots \wedge x'_n = x_n] \\
\vee \\
[OP_2 \mid y'_1 = y_1 \wedge \dots \wedge y'_m = y_m] \vdash
\end{array}
}{\vdash} \quad (\textit{Choice}) [p]$$

$$\begin{aligned}
\text{where } p &\equiv \delta_{OP}(OP_2) \setminus \delta_{OP}(OP_1) = \{x_1, \dots, x_n\} \\
&\quad \delta_{OP}(OP_1) \setminus \delta_{OP}(OP_2) = \{y_1, \dots, y_m\}
\end{aligned}$$

6.4 Enrichment

If in the environment enriched with the schema of OP_1 we can show that the schema of OP_2 is equal to $[d \mid p]$, where the free variables of d do not include any variables declared in OP_1 , then the schema of $OP_1 \bullet OP_2$ is equal to $[OP_1; d \mid p]$.

$$\frac{
OP_1 \vdash OP_2 = [d \mid p]
}{\vdash OP_1 \bullet OP_2 = [OP_1; d \mid p]} \quad (\textit{Enrichment}) [\phi(d) \cap \alpha(OP_1) = \emptyset]$$

For example, consider evaluating the operation expression

$$[s : \mathbb{P}\mathbb{N}] \bullet [x : s]$$

Since the following sequent is valid

$$[s : \mathbb{P}\mathbb{N}] \vdash [x : s] = [x : \mathbb{N} \mid x \in s]$$

and $\phi(x : \mathbb{N}) = \emptyset$, using *Enrichment* we can deduce

$$[s : \mathbb{P}\mathbb{N}] \bullet [x : s] = [s : \mathbb{P}\mathbb{N}; x : \mathbb{N} \mid x \in s]$$

6.5 Renaming

The schema of the operation $OP[\lambda]$ is the schema of OP renamed by λ .

$$\frac{OP[\lambda] = [OP][\lambda] \vdash}{\vdash} \quad (OpRenaming)$$

6.6 Hiding

The schema of the operation $OP \setminus (x_1, \dots, x_n)$ is the schema of OP with x_1, \dots, x_n hidden.

$$\frac{OP \setminus (x_1, \dots, x_n) = [OP] \setminus (x_1, \dots, x_n) \vdash}{\vdash} \quad (OpHiding)$$

7 Objects

To reason about an object of a class, we employ the dot notation to refer to its types, constants, pre-state variables (unprimed) and post-state variables (primed). In addition to the notation $a.x$ of Object-Z, therefore, we introduce the notation $a.T$ to denote the local type T of the class of the object referenced by a and $a.x'$ to denote the post-value of the primary or secondary variable x of the object referenced by a .

7.1 Local types

A local type will be the same for all objects of the class.

$$\frac{\vdash a_1 \in A \quad \vdash a_2 \in A}{\vdash a_1.T = a_2.T} \quad (LocalTypes) [T \in \alpha(A)]$$

7.2 Class membership

A meta-substitution operator \oplus is introduced to enable the substitution of the types, attributes and primed variables of an object into a predicate. It is defined in terms of the meta-substitution operator \odot as follows.

$$a \circledast p = b \odot p$$

where $a \in A$

$$\{x_1, \dots, x_n\} = \alpha(A)$$

$$\{x_k, \dots, x_n\} = \alpha_1(A) \cup \alpha_2(A)$$

$$b = \langle \langle x_1 \rightsquigarrow a.x_1, \dots, x_n \rightsquigarrow a.x_n, x'_k \rightsquigarrow a.x'_k, \dots, x'_n \rightsquigarrow a.x'_n \rangle \rangle$$

Similarly, \circledast is defined for substitution of the types, attributes and primed variables of an object into expressions and declarations.

If an object is a member of a class A then its types, attributes and primed variables satisfy the properties of A 's local environment and the predicate of A 's state schema (in both primed and unprimed form).

$$\frac{\vdash a \in A \quad A :: \vdash STATE = S}{\vdash a \circledast (\eta(A) \wedge \Delta S)} \quad (\textit{ClassMembership}) [\phi(S) \subseteq \alpha(A)]$$

7.3 Initialising objects

The predicate $a.INIT$ is true if and only if the attributes of a satisfy the predicate of A 's initial state schema.

$$\frac{\vdash a \in A \quad A :: \vdash INIT = S}{\vdash a.INIT \Leftrightarrow a \circledast S} \quad (\textit{Initialisation}) [\phi(S) \subseteq \alpha(A)]$$

7.4 Applying operations to objects

The meta-function δ is introduced to return the set of names of variables in the delta-list of an operation of a class. It is defined in terms of a meta-function δ_d which returns the explicitly declared delta-list of the operation.

Given a class $A[X_1, \dots, X_n]$ with operation definition $Op \doteq OP$,

$$\delta_d(A[t_1, \dots, t_n], Op) = \delta_{OP}(OP)$$

To account for the delta-lists of the inherited parts of operations we have

$$\begin{aligned} \delta(A[\lambda], Op) &= \langle \langle \lambda \rangle \rangle \odot \delta(A, Op) \\ \delta(B, Op) &= \delta_d(B, Op) \cup \delta(A_1, Op) \cup \dots \cup \delta(A_k, Op) \quad \text{if } Op \in \Omega_d(B) \\ &= \delta(A_1, Op) \cup \dots \cup \delta(A_k, Op) \quad \text{if } Op \notin \Omega_d(B) \\ \text{where } \iota(B) &= \{A_1, \dots, A_n\} \\ &Op \in \Omega(A_1) \cap \dots \cap \Omega(A_k) \\ &Op \notin \Omega(A_{k+1}) \cup \dots \cup \Omega(A_n) \end{aligned}$$

The schema of the operation $a.Op$ is formed by

- taking the schema of Op of a 's class,
- equating all primary variables not in Op 's delta list with their primed counterparts,

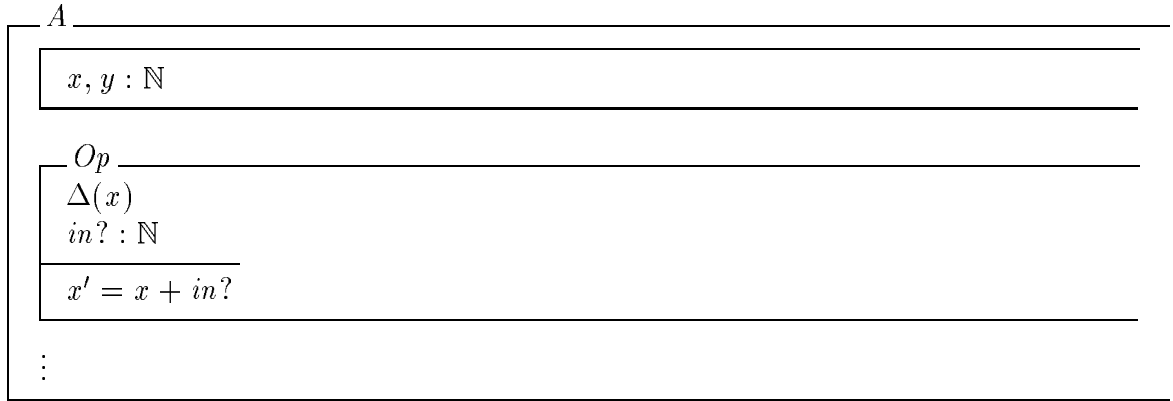
- substituting the types, attributes and primed variables of a into the declaration and the predicate of the resulting schema
- and then hiding the primed and unprimed variables of a 's class (leaving only the variables declared explicitly in Op).

$$\frac{\vdash a \in A \quad A :: \vdash Op = [d \mid p]}{\vdash a.Op = \left[a \otimes d \mid a \otimes (p \wedge x'_1 = x_1 \wedge \dots \wedge x'_n = x_n) \right] \setminus (y_1, \dots, y_m, y'_1, \dots, y'_m)} \quad (OpApplication) [p]$$

where $p \equiv Op \in \Omega(A)$

$$\begin{aligned} \alpha_1(A) \setminus \delta(A, Op) &= \{x_1, \dots, x_n\} \\ \alpha_1(A) \cup \alpha_2(A) &= \{y_1, \dots, y_m\} \\ \phi [d \mid p] &\subseteq \alpha(A) \end{aligned}$$

For example, given the following class



using *StateDef*, *OpDef* and *Enrichment* we can deduce

$$A :: \vdash Op = [x, x', y, y' : \mathbb{N}; in? : \mathbb{N} \mid x' = x + in?]$$

Therefore, given $a : A$, since $\alpha_1(A) = \{x, y\}$, $\alpha_2(A) = \emptyset$ and $\delta(A, Op) = \{x\}$, using *OpApplication* we can deduce

$$\begin{aligned} a.Op &= ([a \otimes (x, x', y, y' : \mathbb{N}; in? : \mathbb{N}) \mid a \otimes (x' = x + in? \wedge y' = y)]) \\ &\quad \setminus (x, x', y, y') \\ &= ([x, x', y, y' : \mathbb{N}; in? : \mathbb{N} \mid a.x' = a.x + in? \wedge a.y' = a.y]) \\ &\quad \setminus (x, x', y, y') \\ &= [in? : \mathbb{N} \mid a.x' = a.x + in? \wedge a.y' = a.y] \end{aligned}$$

7.5 Polymorphism

If a is a member of $\downarrow A$ then a 's class is either A or a class derived from A by inheritance.

$$\frac{\vdash a \in \downarrow A}{\vdash a \in A_1 \vee \dots \vee a \in A_n} \quad (Polymorphism) [(\iota^*)^*(\{A\}) = \{A_1, \dots, A_n\}]$$

8 Conclusion

This paper has presented a logic for Object-Z which extends \mathcal{W} , the logic for Z adopted as the basis of the deductive system in the Z Base Standard. The sequent notation of \mathcal{W} was extended to allow reasoning within the local environments of Object-Z classes and inference rules to cover the additional constructs and operators of Object-Z were presented.

The extended logic forms a basis on which tool support for reasoning about Object-Z specifications can be developed. Preliminary encoding of the rules has begun on the Ergo theorem prover[14]. Its encoding on theorem provers which already support \mathcal{W} (e.g. [9]) should also be feasible.

As well as a basis for reasoning, the logic also provides an abstract, axiomatic semantics of Object-Z formalising many constructs in the language which have only been described informally previously. The logic is yet to be proved sound; however, a denotational semantics of Object-Z is currently being developed and a proof of soundness with respect to this will be carried out when it is completed.

While the logic provides a basis for reasoning about Object-Z specifications, it doesn't explain how such reasoning should proceed. Even the simplest proofs require a sequence of inference rules to be applied which may not always be obvious. Therefore, a practical proof system for Object-Z requires, in addition to the logic, tactics which perform sequences of proof steps corresponding to commonly performed proofs. This represents a major area of future work involving the application of the logic to several case studies to determine the kinds of tactics required.

Acknowledgements

The author would like to thank Roger Duke, Alena Griffiths, Wendy Johnston and Gordon Rose for many helpful discussions. This work is supported by the Australian Research Council (ARC).

References

- [1] Brien S, Nicholls J. Z Base Standard: Version 1. Technical Report PRG-107, Oxford University Computing Laboratory, 1992.
- [2] Diller A. Z: An Introduction to Formal Methods. John Wiley and Sons, 1990.
- [3] Dong J, Duke R. Class Union and Polymorphism. In Mingins C, Haebich W, Potter J, Meyer B (eds), Technology of Object-Oriented Languages and Systems (TOOLS 12 & 9), pp 181–190. Prentice-Hall International, 1993.
- [4] Dong J, Duke R. The Geometry of Object Containment. Technical Report 94-17, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1994. To appear in Object-Oriented Systems (OOS).
- [5] Duke D, Duke R. Towards a semantics for Object-Z. In Bjørner D, Hoare C, Langmaack H (eds), VDM'90: VDM and Z!, vol 428 of Lecture Notes in Computer Science, pp 242–262. Springer-Verlag, 1990.

- [6] Duke R, King P, Rose G, Smith G. The Object-Z specification language: Version 1. Technical Report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1991.
- [7] Duke R, Rose G, Smith G. Object-Z: a specification language advocated for the description of standards. Technical Report 94-45, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1994.
- [8] Griffiths A, Rose G. A semantic foundation for object identity in formal specification. Technical Report 94-21, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1994.
- [9] Martin A. Encoding \mathcal{W} : A logic for Z in 2OBJ. In Woodcock J, Larsen P (eds), FME'93: Industrial-Strength Formal Methods, vol 670 of Lecture Notes in Computer Science, pp 462–481. Springer-Verlag, 1993.
- [10] Rose G. Object-Z. In Stepney S, Barden R, Cooper D (eds), Object Orientation in Z, Workshops in Computing, pp 59–77. Springer-Verlag, 1992.
- [11] Rose G, Duke R. An Object-Z specification of a mobile phone system. In Lano K, Houghton H (eds), Object-Oriented Specification Case Studies, pp 110–129. Prentice-Hall International, 1993.
- [12] Spivey J. Understanding Z: A specification language and its formal semantics, vol 3 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, UK, 1988.
- [13] Spivey J. The Z Notation: A Reference Manual (2nd Ed.). Series in Computer Science. Prentice-Hall International, 1992.
- [14] Utting M, Whitwell K. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1994.
- [15] Woodcock J, Brien S. \mathcal{W} : A logic for Z. In Nicholls J (ed), Z User Workshop, Workshops in Computing, pp 77–98. Springer-Verlag, 1992.
- [16] Woodcock J, Loomes M. Software Engineering Mathematics. Pitman, 1988.
- [17] Wordsworth J. Software Development with Z: A Practical Approach to Formal Methods in Software Engineering. International Computer Science Series. Addison-Wesley, 1992.