Formal Object Modelling Techniques and Denotational Semantics Studies

Jin Song Dong

BInfTech (Honours)

A thesis submitted to

The Department of Computer Science

The University of Queensland

for the degree of

DOCTOR OF PHILOSOPHY

October 1995

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original, except as acknowledged in the text, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

Jin Song Dong

Brisbane, October 1995

Acknowledgement

I am deeply indebted to my supervisor, Dr Roger Duke, for his guidance, insight and encouragement throughout the course of my doctoral program and for his careful reading of and constructive criticisms and suggestions on drafts of this thesis and other works. I am also very grateful to Professor Gordon Rose for his help and guidance throughout my studies.

I owe thanks to Steven Atkinson, Lesley Chase, Alena Griffiths, Andrew Hussey, Wendy Johnston, Anthony MacDonald, Dr Andreas Prinz, Dr Udaya Shankar, Dr Graeme Smith, Dr Paul Stroop and Dr Paul Swatman for many helpful comments and discussions on the issues raised in this thesis. I would also like to thank my friends and office-mates for their help, discussions and friendship. In particular, I would like to thank my best friend Shannen Dan Dan for drawing artistic cartoons for my thesis.

I would also like to thank the numerous anonymous referees who have reviewed parts of this work prior to publication in journals and conference proceedings and whose valuable comments have contributed to the clarification of many of the ideas presented in this thesis.

This project received funding from an Australian Postgraduate Award and from the Department of Computer Science. The Department of Computer Science also provided the finance for me to present papers in several conferences both in Australia and overseas. In addition, I have been encouraged by receiving the Richard Jago Memorial Prize to further my research by a visit to an overseas institution. For all this, I am very grateful. I would also like to acknowledge the Software Verification Research Centre and the Advanced Study Institute (NATO Science Committee) of Germany for financing attendance at the 1994 Marktoberdorf International Summer School.

I sincerely thank my parents YiRong and Xiang, and my aunt Yuan Yuan for their love, encouragement and financial support in my earlier years of study.

Abstract

Formal descriptions of programming languages support language standardisation, program verification and software reliability. State-based formal specification languages such as VDM and Z have been used to define the denotational semantics of programming languages. Usually, the abstract syntax, static semantics and dynamic semantics of a 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. The general lack of modularity and reusability of the traditional denotational semantics representations has been recognised in the literature as a drawback. The main aim of this thesis is to apply an object-oriented approach to specify programming language semantics using the Object-Z notation. The key idea of this approach is to view programming language constructs, such as expressions and statements, as objects. With this approach, the abstract syntax, static semantics and dynamic semantics of an individual language construct are typically defined in one class such that the semantics representation is structural. 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. Furthermore, it is also possible with this approach to reuse parts of the semantics specification of one programming language to define another.

Specifying programming languages in Object-Z not only leads to a more structured language semantic representation but also aids the development of the Object-Z notation itself. For instance, the use of the Object-Z notation to specify programming languages motivates the development of a generalised polymorphic construct, class-union, and the development of notations for capturing object containment in Object-Z. These new extensions to Object-Z not only play important roles in the object-oriented definition of programming languages but also prove to be useful for modelling object-oriented systems in general. This thesis presents these extensions together with the Object-Z specifications of programming languages.

Contents

1 Introduction and Overview **1**

1.1 Motivation and Goals 1

1.2 Thesis Outline and Overview 3

1.2.1 Chapter 2 3

1.2.2 Chapter 3 4

1.2.3 Chapter 4 4

1.2.4 Chapter 5 5

1.2.5 Chapter 6 5

1.2.6 Chapter 7 5

1.2.7 Chapter 8 6

1.2.8 Chapter 9 7

1.2.9 Chapter 10 7

1.2.10 Chapter 11 7

1.3 Publications From the Thesis 8

2 Z and Object-Z Background **9**

2.1 Z Overview 9

2.2 Free Types in Z 10

2.3 Object-Z Overview 12

2.3.1 Class 12

2.3.2 Inheritance 14

2.3.3 The Use of Generic Classes 16

2.3.4 Instantiation 17

2.3.5 Polymorphism 19

3 The Role of Secondary Attributes **21**

3.1 Secondary Attributes and Delta Lists 23

3.1.1 A Hotel Management System 23

3.2 Adding Clarity in Specification 25

3.2.1 A Complex Variable 25

3.3 Dependency on Environment: Sharing 28

3.3.1 A Game Sharing Group 28

3.4 Dependency Chains and Recursion 32

3.4.1 A Simple Predicates Store 32

3.5 Conclusion 37

4 An OO Approach to the Semantics of Programming Languages **39**

4.1 Semantics of Expressions in Z and Object-Z 40

4.1.1 Concrete Syntax of Expressions 41

4.1.2 Syntax and Semantics of Expressions in Z 41

4.1.3 Syntax and Semantics of Expressions in Object-Z 43

4.2 Language Semantics in Object-Z 47

4.2.1 Concrete Syntax 47

4.2.2 Expressions 48

4.2.3 Statements 49

4.2.4 The Program 52

4.3 Discussion 53

4.3.1 Polymorphic Declarations 53

4.3.2 Common Object Reference Structures 54

5 Polymorphic Class Types **55**

5.1 Polymorphic References 56

5.1.1 Example: A Power Tool 56

5.1.2 Example: Ginger Meggs 59

5.1.3 Example: A Channel 61

5.2 Class Union 62

5.2.1 Polymorphic Core 64

5.2.2 Typing 65

5.2.3 Examples Revisited 66

5.2.4 Extending Class Unions 68

5.3 Class Union and Traditional Polymorphism 68

5.4 Case Study: A Telephone System 70

5.4.1 Informal View of the System 70

5.4.2 Specification of the System 71

5.5 Conclusion 76

6 Free Type and Class Union 79

6.1 Polymorphic Structures 80

6.1.1 Modelling Bank Notes in Free Type 81

6.1.2 Modelling Bank Notes in Class Union 82

6.2 Construction of a Polymorphic Core 85

6.3 Recursive Structures 87

6.4 Consistency of Definitions 92

6.5 Conclusion 95

7 Object Containment 97

7.1 Capturing the Properties of Containment 99

7.1.1 Example: Terminal Location 101

7.1.2 Example: Russian Dolls 103

7.2 Capturing the Geometry of Containment 104

7.2.1 Examples Revisited 106

7.2.2 A Notational Simplification 107

7.2.3 Indirectly Contained Objects 108

7.2.4 Object Relocation 109

7.3 Containment in Programming Languages 111

7.4 Containment and Access 113

7.4.1 Example: A Banking System 114

7.5 Modelling Shared Containment 116

7.5.1 Example: Walls, Rooms and Buildings 119

7.5.2 Example: Buildings, Streets and Suburbs 120

7.5.3 Other Containment Geometries 120

7.6 Containment in Abstract Structures 121

7.7 Conclusion 122

8 Exclusive Object Control within Object Oriented Systems 125

8.1 Capturing Exclusive Control: the Problem 126

8.1.1 Example: Piggy-bank 126

8.2 Notation for Exclusive Control 129

8.2.1 Formalising Exclusive Control 130

8.2.2 Object-Z Notation for Exclusive Control 130

8.2.3 Preventing Indirect Access 133

8.3 Case Study: Files and Users 135

8.4 Conclusion and Discussion 138

9 A Block Structured Procedural Language Definition 141

9.1 Concrete Syntax of the Language 142

9.2 Semantics of the Language 142

9.2.1 Expressions 142

9.2.2 Statements 145

9.2.3 Procedure and Program 149

9.3 Conclusion 152

10 Modelling Object Oriented Programming Languages 155

10.1 An OO Approach to Modelling OOPLs 156

10.2 An Object-Oriented Model of an OOPL 158

10.2.1 SOPL and Its Concrete Syntax 158

10.2.2 Modelling Identifiers, Objects and Variable References 160

10.2.3 Modelling Expressions 163

10.2.4 Modelling Statements 166

10.2.5 Modelling Classes 171

10.2.6 Programs 175

10.3 The Ease of Extendibility 176

10.3.1 Adding More Language Constructs 176

10.3.2 Method Redefinition in Class Inheritance 177

10.4 Conclusion 178

11 Conclusions and Directions for Further Research 181

11.1 Thesis Main Contributions and Influence 181

11.2 Directions for Further Research 182

11.2.1 A Standard Object-Z Generic Class Library 182

11.2.2 Applying Secondary Attributes to Specify Distributed Systems 183

11.2.3 Calculating Polymorphic Cores: Type Checking 183

11.2.4 Converting Containment and Exclusive Control Notations into Predicates 183

11.2.5 Guidelines for Applying Containment and Control Notations . 184

11.2.6 Specifying Safety and Security Critical Systems 185

11.2.7 Semantics of Other Programming Language Paradigms 185

A Glossary of Z Notation **199**

Index of Definitions **217**

Chapter 1

Introduction and Overview

1.1 Motivation and Goals

Formal descriptions of programming languages support language standardisation, program verification and software reliability. There are three distinct approaches to specify a programming language — operational semantics, axiomatic semantics and denotational semantics. Operational semantics[63, 82, 93] defines a language in terms of the operation of an abstract machine, which gives an abstract interpreter of the language. Axiomatic semantics[47, 65] defines a language by providing assertions and inference rules for reasoning about programs. The axiomatic specification of a programming language will be a mathematical theory for that language. A denotational semantics description of a programming language defines the meaning of the language in terms of mathematical objects[51, 107, 111]. For studying language concepts and relating implementations to specifications, it is widely accepted that a denotational semantics is most useful[71]. Denotational semantics of the well-known programming languages Ada, Pascal and Lisp can be found in [8, 3, 50]. The formal specification language VDM[70] is often used as a meta-language to describe the semantics of programming languages[7]. Recently the Z[104] notation has similarly been used to define the semantics of programming languages[6, 41, 58, 105]. In addition, Meyer used a mixture of VDM and Z notations to specify programming languages in [84]. The approaches of VDM and Z are very similar (see [58] for a comparison of the two languages[1] when applied to the specification of the semantics of the small Pascal-like procedural language defined in [71]). 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.

However, 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. Therefore the semantic representation of the programming language is not extendible. In addition to this observation, the general lack of modularity and reusability of the traditional denotation semantics has been reconised as a drawback in the literature[89]. However (to our knowledge), no research has been done to overcome this drawback. The main aim of this thesis is to apply an object-oriented approach to specify programming language semantics using the Object-Z[18, 42, 46, 96] notation. With this approach, the abstract syntax, static semantics and dynamic semantics of an individual language construct can be grouped together and captured by a single class construct, so that the structured formal semantics of a programming language can be readily extended when the language is enhanced. Also, reusing parts of the denotational description of one language to define another language is possible with our approach.

Using a formal notation to specify programming languages not only supports standardisation of programming languages and compiler design but also aids the development of the formal notation itself. As Jones states,

[1] More general discussions on the difference of Z and VDM notations can be found in [59, 60].

> 'The main stimulus for the inception of VDM was the description of programming languages.' ([71] p 235)

Although the Object-Z specification language has been widely used to specify communication systems[43, 45, 79, 97], open distributed processing (ODP) systems[13, 30, 106], information systems[73, 110, 109] and others[37, 81], it has not been used for the specification of programming languages. Using Object-Z to specify programming languages provides useful feedback and enriches the Object-Z specification language itself. Therefore two goals of this project are

- to apply an object-oriented approach to specify programming languages using the Object-Z notation so that the semantics representations of programming languages are well structured, extendible and reusable;
- to further develop and enhance the Object-Z specification language.

1.2 Thesis Outline and Overview

The structure of the thesis is as follows.

1.2.1 Chapter 2

This chapter is devoted to an overview of the Z notation and the Object-Z notation. Most constructs of Z are fairly standard mathematical notations and in this chapter we only briefly outline the Z notation. However, a special Z construct, free type (disjoint union)[104], needs a more detailed explanation because it plays an important role in specifying semantics of programming languages in Chapter 4 and Chapter 9. In Chapter 6, free type will be further discussed and compared to the class-union construct (introduced in Chapter 5).

Object-Z is an object-oriented extension to Z. The main object-oriented features, such as class construct, inheritance, polymorphism and instantiation in Object-Z are briefly discussed in this chapter. The use of generic classes as a class library to specify object-oriented systems is also addressed.

1.2.2 Chapter 3

When modelling a large and complex system, such as the semantics of a programming language, clarity of the specification becomes an important factor. In object-oriented specification, the states of individual objects are captured by the values of their attributes. Frequently however, there are dependencies between the attributes of an object. An appropriate indication of which attributes are primary (independent) and which are secondary (dependent) can add significantly to clarity. Chapter 3 details the notion of secondary attributes, their roles and implications in formal object-oriented specification. Secondary attributes are also applied frequently and extensively throughout the remaining chapters of the thesis.

1.2.3 Chapter 4

This chapter presents an object-oriented approach to specify the semantics of a very simple procedural programming language using the Object-Z notation. The key idea of this approach is to view programming language constructs, such as expressions and statements, as objects. In this chapter the semantics of simple expressions of the simple programming language is specified in both Z and Object-Z and the resulting formal structures compared. Also in this chapter, I identify and discuss some weakness of this early version of the object-oriented specification of the programming language and these discussions motivate the development of a generalised polymorphic construct, class-union, and the development of notations for capturing object containment in Object-Z. These new extensions to Object-Z not only play important roles in the object-oriented definition of programming languages but also prove to be useful for modelling object-oriented systems in general. Therefore in Chapter 5 and 7, I present these extensions with the aim of modelling object-oriented systems in general rather than being concerned only with the programming language model.

1.2.4 Chapter 5

This chapter presents the development of the class-union construct in Object-Z. Polymorphism within object-oriented systems has traditionally been restricted to suitable inheritance hierarchies. This restriction arose historically and has persisted for pragmatic reasons despite the fact that it has long been recognised that polymorphism and inheritance are orthogonal concepts. In this chapter some examples are considered that suggest a form of polymorphism not related to inheritance. Furthermore, these examples illustrate that polymorphism can have some meaning even in cases where the underlying classes are not subtypes and have little, if any, behavioural compatibility. The specification of such systems is facilitated by a class-union construct; this construct is defined and its implications explored. Class-union construct will play an important role for specifying the semantics of programming languages in Chapters 9 and 10.

1.2.5 Chapter 6

The free type construct (outlined in Chapter 2) and the class-union construct (introduced in Chapter 5) can both be used for modelling polymorphic and recursive structures. However, free type and class-union constructs are respectively based upon the functional value point of view and the object reference point of view. Consequently, the roles these two constructs perform in system modelling are different. This chapter compares and discusses the free type and class-union constructs through various examples. The aim of this comparison and discussion is to present guidelines on how appropriately and effectively to use these two constructs to specify polymorphic and recursive structures.

1.2.6 Chapter 7

This chapter presents the development of the notation for capturing object containment in Object-Z. In object-oriented systems, it is often the case that an object will have an attribute whose value identifies (points or refers to) some other object in the system so that the identified object can be sent messages. The association between objects determined by the object references in a system will generally result in a complex structure whose design and specification is a crucial part of the development and implementation of the system. In this chapter, we look at ways of capturing formally object reference structures that occur frequently in object-oriented systems. For example, consider a system consisting of car and wheel objects where each car has attributes referencing its wheel objects. If wheels are not shared between cars, distinct car objects will reference distinct wheel objects. We distinguish such object references by saying that a car object (directly) *contains* the wheel objects it references. The nature of the contained object references as suggested by this example is captured within a formal framework and incorporated into the Object-Z specification language. Object containment is then compared with related ideas found in object-oriented programming languages. Furthermore, the distinction between the notions of object containment and object access is discussed, which facilitates the development of the notation for exclusive object control in Chapter 8. Finally, a generalised notion of object containment is also investigated. The object containment notation will play an important role for specifying the semantics of programming languages in Chapter 9 and 10.

1.2.7 Chapter 8

The notion of object containment (developed in Chapter 7) is concerned only with the relative geometric patterns of some common object associations; it is not concerned with the access control aspect of object associations. However, a particularly common object association arises when one object exclusively controls (owns) another. As exclusive object control is an important aspect of safety and security critical systems, specific notation to capture directly such a notion needs to be included in any formal specification language. This chapter first discusses the problems that arise when using the Object-Z notations, such as object containment, to capture the notion of exclusive control. Then the Object-Z notation is extended to enable the notion of exclusive control to be modelled directly. Finally, a case study is presented to contrast the distinction between the use of exclusive control and object containment when modelling object-oriented systems.

1.2.8 Chapter 9

This chapter applies the new extensions (class-union and object containment) to present the specification of a block structured programming language. In this chapter, the class-union construct is used to capture abstractly the recursive and polymorphic nature of the language and the notation of object containment is used to capture the object reference structure of the programming language constructs. By comparison with the specification in Chapter 3, the advantage of applying class-union and object containment to specify the semantics of programming language is demonstrated. Furthermore, the block structured programming language includes procedures with both value and reference parameter substitution which adds a level of complexity to the language semantic representation.

1.2.9 Chapter 10

This chapter applies the object-oriented approach to investigate the semantics of the object-oriented programming language paradigm. Not only is each language construct of an object-oriented programming language, such as expressions, statements, methods and classes modelled by Object-Z classes, but also even run-time entities of an object-oriented program such as objects (instances of classes), are modelled by Object-Z classes. In this chapter, the concepts of object-orientation, such as the relationship between 'class' and 'object', 'inheritance' and 'polymorphism', are also specified in the Object-Z model.

1.2.10 Chapter 11

Chapter 11 concludes the thesis with a summary of the main contributions of this thesis, and some suggestions for further research.

1.3 Publications From the Thesis

Most chapters of the thesis have been accepted in journals or conference proceedings. Parts of the work in Chapter 2 were used as a basis for the paper presented at *The International Conference on Open Distributed Processing ICODP'93 (Sep 93, Berlin)*[30]. Chapter 3 was presented at the *First IEEE International Conference on Engineering Complex Computer Systems ICECCS'95 (Nov 95, Florida)*[36]. Chapter 4 was presented at *The 17th Annual Computer Science Conference ACSC'17 (Jan 94, Christchurch)*[34]. Chapter 5 was presented at *The 12th International Conference on Technology of Object-Oriented Languages and Systems TOOLS'12 (Nov 93, Melbourne)*[29]. Chapter 6 was presented at *The 1995 Asia Pacific Software Enginering Conference*[28]. Chapter 7 has been published in the first issue of the second volume of the *Object-Oriented Systems (OOS)* journal[32]. Chapter 8 was presented at *The 18th International Conference on Technology of Object-Oriented Languages and Systems TOOLS'18 (Nov 95, Melbourne)*[31]. Chapters 9 has been accepted by *The 20th International Conference on Technology of Object-Oriented Languages and Systems TOOLS'20 (July, Santa Barbara)*[33]. Chapter 10[35] have been submitted for publication.

Chapter 2

Z and Object-Z Background

This chapter reviews the Z and the Object-Z notation. The free type construct in Z and the various object-oriented features in Object-Z are discussed in detail; in particular, the use of generic classes for defining specific classes in Object-Z is addressed.

2.1 Z Overview

The Z notation[104] is a state-oriented formal specification language based on set theory and predicate calculus. It was developed by the Programming Research Group at Oxford University. Z is classified as a model-based specification language (like VDM). It is worth mentioning that there are other styles of formal specifications. For instance:

- Algebraic specifications model systems constructively in terms of applications of system operations. Examples of this style are Clear[14] and OBJ[48].
- Process algebras model systems in terms of behaviour and interaction of agents. Examples of this style are CCS[88], CSP[66] and LOTOS[10].

A Z specification typically includes a number of state and operation schema definitions. A state schema encapsulates variable declarations and related predicates (invariants). The system state is determined by values taken by variables subject to restrictions imposed by state invariants. An operation schema defines the relationship between the 'before' and 'after' states corresponding to one or more state schemas. A schema calculus facilitates the composition of complex specifications. Z has been widely adopted to specify a range of software systems (see [57]). The formal semantics of Z has also been developed[103, 12]. A number of textbooks on Z are also available, e.g. [104, 94, 116, 26]. Most constructs of Z use fairly standard mathematical notation; we shall assume the reader is familiar with the basics of Z (the Z glossary in [57] is provided in Appendix A). A particular Z construct, free type (disjoint union), needs a more detailed explanation because it is applied frequently in the thesis but not so frequently elsewhere.

2.2 Free Types in Z

The Z free type construct[104] is a way of combining two defined sets into a new set. As Z is based on typed set theory, it is invalid to write:

$$Value == \mathbb{B} \cup \mathbb{Z} \cup \mathbb{C}$$

where $\mathbb{B}$, $\mathbb{Z}$ and $\mathbb{C}$ denote booleans, integers and characters respectively. In Z, a more elaborate approach (free type definition) is used to model the above situation:

$$Value ::= bool\langle\!\langle \mathbb{B} \rangle\!\rangle \mid int\langle\!\langle \mathbb{Z} \rangle\!\rangle \mid char\langle\!\langle \mathbb{C} \rangle\!\rangle$$

In the above free type definition, the *bool*, *int* and *char* constructors are injections (total one-to-one functions) from the sets $\mathbb{B}$, $\mathbb{Z}$ and $\mathbb{C}$ to *Value*.

A consequence of using free types is that disjoint sets can be labeled by the names of constructors and grouped together. For instance, an element, '3', of $\mathbb{Z}$ (integers) corresponds to an element, '$int(3)$', of *Value*; an element of *Value*, say '*val*' which is in the range of *int*, corresponds to an element '$int^{\sim}(val)$' of $\mathbb{Z}$ (where the super-script $^{\sim}$ denotes function inverse).

Free type definitions can be recursive and therefore recursive structures can be represented by the free type definitions. For example, a simple linked-list can be specified in Z by using the free type definition as

$$LinkedList ::= nil \mid node\langle\langle \mathbb{Z} \times LinkedList \rangle\rangle$$

This introduces a new type *LinkedList*, a constant *nil* of type *LinkedList* and an injective constructor function *node*, that, given an integer and a linked-list, returns a linked-list. For example, elements 'nil', '$node(3, nil)$', '$node(1, node(3, nil))$' etc are members of *LinkedList*.

A free type definition can be transferred into other more primitive Z constructs, namely given types and a set of constructor functions (axiomatic descriptions). For instance, the free type *LinkedList* can be transferred into the following definitions:

$$[LinkedList]$$

$$\begin{array}{|l}
nil : LinkedList \\
node : (\mathbb{Z} \times LinkedList) \rightarrowtail LinkedList \\
\hline
\mathbf{disjoint}\langle \{nil\}, \operatorname{ran} node \rangle \\
\forall W : \mathbb{P}\, LinkedList \bullet \{nil\} \cup node(\!|\, \mathbb{Z} \times W \,|\!) \subseteq W \Rightarrow LinkedList \subseteq W
\end{array}$$

In the above axiomatic definition, the first predicate constrains the range of *node* so that it does not contain the constant *nil*. The second predicate constrains the set *Linklist* to be closed under the constructor *node*. As the domain of *node*, $\mathbb{Z} \times LinkedList$, is finitary[1], the two predicates implies

$$\langle \{nil\}, \operatorname{ran} node \rangle \ \mathbf{partition}\ LinkedList$$

For a detailed discussion of the theoretical issues, such as the issue of consistency of free types, see [4, 99].

The free type construct is used in Chapter 4 to define the abstract syntax of simple expressions in Z. In Chapter 5, the free type construct is compared to the class-union construct.

[1]Examples of finitary constructions include finite sets $\mathbb{F}\,S$, Cartesian products $S \times T$, finite functions $S \twoheadrightarrow\!\!\!\!\!\!\!\!\;\shortmid\;\; T$, etc and any composition of finitary constructions.

2.3 Object-Z Overview

Object-Z is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. The Object-Z specification language has been developed at the University of Queensland. There are other object-oriented extensions to Z and VDM, such as MooZ[83], OOZE[1], Z++[76], Fresco[113] etc. Among them, Object-Z is regarded by some as the most mature[77]. The semantics of Object-Z have also been developed in [39, 40, 54, 55, 100, 102] [2].

2.3.1 Class

The essential extension to Z in Object-Z is the *class* construct which groups the definition of a state schema and the definitions of its associated operations. A class is a template for *objects* of that class: for each such object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class.

Syntactically, a class definition is a named box, optionally with generic parameters. In this box the constituents of the class are defined and related. The main constituents are: a state schema, an initial state schema and operation schemas.

$$
\begin{array}{|l}
\hline
\textit{ClassName}[\textit{generic parameters}] \\
\hline
\textit{inherited classes} \\
\textit{state schema} \\
\textit{initial state schema} \\
\textit{operation schemas} \\
\hline
\end{array}
$$

[2]The semantics foundations for object identity[44] has been developed in [54, 55], where the work is related to the Z base standard project [12]. Axiomatic semantics of Object-Z has been developed in [102] by extending the logic $\mathcal{W}$[115].

A generic collection example

Consider the following specification of the generic class *Collection*$[T]$ which denotes a collection of elements of T. The class contains operations to add elements to, and delete elements from, the collection.

Collection$[T]$

$$max : \mathbb{N}$$
$$elemts : \mathbb{P}\, T$$
$$\#elemts \leq max$$

INIT

$$elemts = \varnothing$$

Add

$$\Delta(elemts)$$
$$x? : T$$
$$x? \notin elemts$$
$$elemts' = elemts \cup \{x?\}$$

Delete

$$\Delta(elemts)$$
$$x! : T$$
$$x! \in elemts$$
$$elemts' = elemts \setminus \{x!\}$$

The state schema is nameless and contains declarations (the attributes) above the short dividing line and a predicate (class invariant) below the line. In this example, it has one attribute *elemts* denoting a set of elements of the generic type T. The class invariant stipulates that the size of the set cannot exceed the number *max*.

The initial state schema is distinguished by the keyword INIT. The state schema is implicitly included in the initial state schema. In this example, an initialised collection contains no elements of T (i.e. *elemts* is the empty set).

The remaining two schemas are operation schemas. Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means no attribute changes value (further discussion on Δ-lists can be found in Chapter 3).

If an attribute does not appear in the Δ-list of any operation of a class, then the attribute is a constant. In the class $Collection[T]$, the attribute max is a constant. Every operation schema implicitly includes the state schema in un-primed form (the state before the operation) and primed form (the state after the operation). Hence the class invariant holds at all times: in each possible initial state and before and after each operation.

In this example, operation Add adds a given input $x?$ to the existing set provided the set has not already reached its maximum size (an identifier ending in '?' denotes an input). Operation $Delete$ outputs a value $e!$ defined as one element of $elemts$ and reduces $elemts$ by deleting $e!$ from the original set (an identifier ending in '!' denotes an output).

2.3.2 Inheritance

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes.

Essentially, all definitions are pooled with the following provisions. Inherited type and constant definitions and those declared in the derived class are merged. The state and initialisation schemas of inherited classes and those declared in the derived class are conjoined. Operation schemas with the same name are also conjoined.

Inheritance in Object-Z can be used to define a new class by extending an existing class. For instance, the generic class $DAG[T]$ denoting a directed acyclic graph can be defined by inheriting $Collection[T]$.

$DAG[T]$

$Collection[T]$

$linkto : T \leftrightarrow T$

$\mathrm{dom}\, linkto \cup \mathrm{ran}\, linkto \subseteq elemts$

$\nexists e : elemts \bullet (e, e) \in linkto^{+}$

$Delete$

$\Delta(linkto)$

$linkto' = \{x!\} \lhd linkto \rhd \{x!\}$

$AddLink$

$\Delta(linkto)$

$from?, to? : T$

$\{from?, to?\} \subseteq elemts$

$linkto' = linkto \cup \{(from?, to?)\}$

$DeleteLink$

$\Delta(linkto)$

$from?, to? : T$

$\{(from?, to?)\} \in linkto$

$linkto' = linkto \setminus \{(from?, to?)\}$

The class $DAG[T]$ inherits the state variable *elemts* and the operation *Add* from $Collection[T]$. It also includes explicitly the state variable *linkto* denoting the links between members of *elemts* and the extra operations *AddLink* and *DeleteLink*. The operation *Delete* for $DAG[T]$ is defined as the conjunction of the operation *Delete* inherited from $Collection[T]$ and the operation *Delete* declared explicitly in $DAG[T]$.

Inheritance in Object-Z can be also used to add constraints to the state schema of a derived class. For instance, a generic *Forest* class can be defined by inheriting the

DAG class.

$$\begin{array}{|l}
\hline
\textit{Forest}[T] \\
DAG[T] \\
\hline
\textit{linkto}^{\sim} \in T \rightarrowtail\!\!\!\!\!\!\!\!+ \; T \\
\hline
\end{array}$$

The additional state invariant $linkto^{\sim} \in T \nrightarrow T$ of the class $Forest[T]$ ensures that no member of $elemts$ is linked by more than one member (parent) of $elemts$.

2.3.3 The Use of Generic Classes

The Object-Z classes presented so far are all generic classes. A generic class can be instantiated by substituting a specific type for its generic type parameter to define a specific class. For instance, a forest of integers can be modelled as:

$$IntegerForest \mathrel{\widehat=} Forest[\mathbb{Z}]$$

Generic classes are particularly useful for class reuse. In [30], we applied some generic classes defined in this chapter, such as $Collection[T]$ and $DAG[T]$, as a small class library to specify the Open Distributed Processing (ODP) Trader[69]. For instance, the *context manager* of the Trader can be modelled as an aggregate of *context* with hierarchical relations between these contexts, i.e.

$$\begin{array}{|l}
\hline
\textit{ContextManager} \\
DAG[\textit{Context}]\ldots \\
\ldots \\
\hline
\end{array}$$

where $Context$ (an aggregate of $exporters$) is modelled as:

$$\begin{array}{|l}
\hline
\textit{Context} \\
\textit{Collection}[\textit{Exporter}]\ldots \\
\ldots \\
\hline
\end{array}$$

where $Exporter$ represents advertising agencies in ODP systems. For the detailed specification of the ODP Trader in Object-Z, see [30].

2.3.4 Instantiation

If A is the name of a class, the identifier A semantically also denotes the set of identities of possible objects of class A. (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.) Informally, we do not distinguish between an object and its identity, i.e. if we refer to some *object* of A, it is with the understanding that we are in fact referring to an object whose identity is in A. Because object identities uniquely identify objects, no ambiguity arises.

Objects may have object references as attributes, i.e. conceptually, an object may have constituent objects. Such references may either be individually named or occur in aggregates. For example, the declaration $c : C$ declares c to be a reference to an object of the class described by C. A declaration $c, d : C$ need not mean that c and d reference distinct objects. If the intention is that they do so at all times, then the predicate $c \neq d$ would be included in the class invariant.

The term $c.att$ denotes the value of attribute att of the object referenced by c, and $c.Op$ denotes the evolution of the object according to the definition of Op in the class C.

Example: Two Forests

Suppose we want to model an aggregate of two forests (f_1 and f_2) with operations to add an element into a given forest f_1 or f_2, to delete an element from f_1 or f_2 (nondeterministically selected), to transfer an element from f_1 to f_2 and to relocate all the elements of f_1 into f_2 one by one. (One aim of this example is to demonstrate the use of different kinds of operators in Object-Z.)

$TwoForests[T]$

$f_1, f_2 : Forest[T]$

$f_1.elemts \cap f_2.elemts = \varnothing$

INIT

$f_1.\text{INIT} \wedge f_2.\text{INIT}$

$SelectForest$

$f? : Forest[T]$

$(f? = f_1) \vee (f? = f_2)$

$AddOne \mathrel{\widehat=} SelectForest \bullet f?.Add$

$DeleteOne \mathrel{\widehat=} f_1.Delete \;[]\; f_2.Delete$

$Transfer \mathrel{\widehat=} f_1.Delete \parallel f_2.Add$

$$RelocateAll \mathrel{\widehat=} Transfer \fatsemi RelocateAll \;[]\; [f_1.elemts = \langle\,\rangle \vee \#f_2.elemts = f_2.max]$$

The declaration $f_1, f_2 : Forest[T]$ models an aggregate of two forest objects.

The evolution of a constituent forest object f_1 or f_2 in the aggregate is affected by defining a selection environment such as *SelectForest* that selects a forest (f_1 or f_2); the operation $AddOne$ is then applied to the chosen object. (The notation $schema_1 \bullet schema_2$ means that variables declared in the signature of $schema_1$ are accessible when interpreting $schema_2$.)

The choice operator ' $[]$ ' used in the definition of $DeleteOne$ indicates non-deterministic choice of one component operation from those component operations with satisfied preconditions.

The parallel operator '$\parallel$' used in the definition of *Transfer* achieves inter-object communication: the operator conjoins constraints and equates variables with the same name and also equates and hides any input variable to one of the components of $\|$ with any output from the other component that has the same basename (i.e. the inputs and outputs are denoted by the same identifier apart from ? and ! decorations)

The sequential composition operator '$\fatsemi$' used in the definition of *RelocateAll* is similar to the sequential composition operator defined in the Z notation; it behaves like forward relational composition. The operation *RelocateAll* is recursively defined. The recursion is a relational composition chain of *Transfer* operations terminated by the operation

$$[\, f_1.elemts = \langle\rangle \vee \# f_2.elemts = 100 \,] \, .$$

The operator '$[]$' in the operation *RelocateAll* behaves deterministically as exactly one precondition of the two branch operations is true.

The formal semantics of the Object-Z operators, such as '$\wedge$', '$\|$', '$[]$' and '$\fatsemi$' (including the recursive operations) are formally defined in [101, 102].

2.3.5 Polymorphism

In Object-Z, the notation $a : \downarrow A$ means that a is an identity of an object in class A or in any subclasses (by inheritance) of A. For instance, declaration $x : \downarrow Collection[T]$ introduces x as a reference to an object of class $Collection[T]$ or any (direct or indirect) derivative of $Collection[T]$, such as $DAG[T]$ and $Forest[T]$.

Another mechanism for polymorphism in Object-Z is the *class union* construct [29], which explicitly lists a collection of classes. Class union and polymorphism will be discussed in detail in Chapter 5.

An overview of Z and Object-Z has been presented in this chapter. The following chapters present the main contributions of this thesis.

Chapter 3

The Role of Secondary Attributes

When specifying an object, its state is captured by the values of its attributes and its potential behaviour is determined by the specification of its operation. In general, there are many ways to model an object. For example, the shape of a triangle (Figure 3.1) can be modelled as an object with three attributes representing either (the lengths of) the three sides of the triangle, or two sides and (the size of) their included angle, or two angles and the side between the two angles. These three alternatives are specified in three skeletal classes, $Triangle_{SSS}$, $Triangle_{SAS}$ and $Triangle_{ASA}$ using the Object-Z specification language.

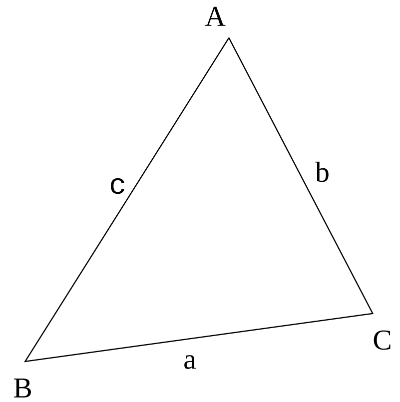


Figure 3.1: Triangle ABC.

$$\begin{array}{|l}\hline \textit{Triangle}_{SSS} \\ \hline a, b, c : \mathbb{R}^+ \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{Triangle}_{SAS} \\ \hline a, B, c : \mathbb{R}^+ \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{Triangle}_{ASA} \\ \hline A, b, C : \mathbb{R}^+ \\ \hline \end{array}$$

$\mathbb{R}^+$ represents the positive, non-zero real numbers. The three representations all use the minimum number of attributes to uniquely capture the shape of a triangle. However preference depends on context. If all conventional aspects of a triangle need to be represented then a triangle would be modelled as follows:

$$\begin{array}{|l}\hline \textit{Triangle} \\ \hline a, b, c, A, B, C, \textit{area}, \textit{perimeter} : \mathbb{R}^+ \\ \hline \end{array}$$

Clearly, there are many dependencies between these eight attributes,
e.g. $perimeter = a + b + c$ and $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)}$. In general, it is debatable which attributes are more important. However, given a specific system, there is normally an implied preference and an explicit indication of this preference can add clarity to the specification. In large and complex systems, clarity becomes particularly important. Rumbaugh[98] suggests that the contents of an object should be separated into two parts — *base* (primary) information and *derived* (secondary) information for clarity representation. For instance, if the sides of a triangle are given preference, then attributes a, b, c of *Triangle* become the base attributes and other attributes become derived. The notion of secondary attributes has also been realised in [44], where

a concept similar to Rumbaugh's derived attribute, namely *dependent variable*, is introduced into the Object-Z. Although Rumbaugh's introduction of the concept of derived information is informal, it provides a good starting point for discussion of the notion of secondary attributes in Object-Z. This thesis clarifies the roles and explores the implications of secondary attributes in formal modelling.

The structure of this chapter is as follows. Section 3.1 discusses the definition of secondary attributes using a simple hotel management system as an example. Section 3.2 illustrates how secondary attributes can simplify and clarify a specification. Section 3.3 shows the use of a secondary attribute in modelling object sharing, and demonstrates that the value of a secondary attribute may depend on the environment rather than on the primary attributes of the object itself. Section 3.4 presents the role of the secondary attributes in modelling recursively structured objects.

3.1 Secondary Attributes and Delta Lists

In this section, we use the specification of a simple hotel management system to illustrate the semantic relationship between secondary attributes and Δ-lists.

3.1.1 A Hotel Management System

Consider a simple hotel management system. Suppose the system stores information on rooms and their occupancy (total number of rooms in the hotel is 100). A client may *CheckIn* to an unoccupied room or *CheckOut* of an occupied room. Vacant rooms can be listed at any time (*ListVacancies*).

Let $[Room]$ represent the type of a room in a hotel. This simple system can be modelled in Object-Z as:

HotelManagementSystem

$rooms : \mathbb{P}\, Room$
$occupied : \mathbb{P}\, Room$
Δ
$vacant : \mathbb{P}\, Room$

$occupied \subseteq rooms$
$vacant = rooms \setminus occupied$

INIT

$\# rooms = 100$
$occupied = \varnothing$

CheckIn

$\Delta(occupied)$
$x? : Room$

$x? \in vacant$
$occupied' = occupied \cup \{x?\}$

CheckOut

$\Delta(occupied)$
$x? : Room$

$x? \in occupied$
$occupied' = occupied \setminus \{x?\}$

ListVacancies

$rooms! : \mathbb{P}\, Room$

$rooms! = vacant$

The primary attributes are *rooms* and *occupied* while the only secondary attribute is *vacant*. The dependency of the secondary attribute *vacant* on primary attributes *rooms* and *occupied* is specified in the second conjunct of the class invariant. In Object-Z, the declaration of primary and secondary attributes are syntactically separated by the Δ symbol, which indicates that secondary attributes are implicitly included in the Δ-list of every operation. The understanding of the Δ-list of an operation in Object-Z is that attributes which are so listed are subject to change. Therefore secondary attributes are always subject to change whenever any operation is invoked. Primary attributes not included in the Δ-list of an operation are implicitly unchanged on application of the operation. For instance, if operation *CheckIn* is applied, the predicate $rooms' = rooms$ is implicitly included, whereas the other attributes, *occupied* and *vacant*, are subject to change. The changes are specified by the predicate of *CheckIn* and the class invariant. Note that even though the secondary attribute *vacant* is implicitly included in the Δ-list of operation *ListVacancies*, it remains unchanged because both attributes, *rooms* and *occupied*, being unlisted are unchanged and *vacant* is functionally determined by them.

Notice that the *HotelManagementSystem* class can be defined by using the generic class $Collection[T]$ (appeared in Chapter 2) with appropriate renaming, i.e.

HotelManagementSystem

$Collection[Room][occupied/elemts, CheckIn/Add, CheckOut/Delete]$

$rooms : \mathbb{P}\, Room$
Δ
$vacant : \mathbb{P}\, Room$

$occupied \subseteq rooms$
$vacant = rooms \setminus occupied$

INIT

$max = 100$
$\#rooms = max$

ListVacancies

$rooms! : \mathbb{P}\, Room$

$rooms! = vacant$

The *HotelManagementSystem* example has illustrated the underlying semantics of a secondary attribute in terms of the Δ-list. In the following sections, the different roles of secondary attributes in formal system modelling are demonstrated.

3.2 Adding Clarity in Specification

In this section, we use the specification of a complex number variable to illustrate how secondary attributes can simplify and clarify a specification.

3.2.1 A Complex Variable

Consider a complex variable with two component attributes, a real part and an imaginary part, and suppose the variable can be changed by the following few operations (Figure 3.2):

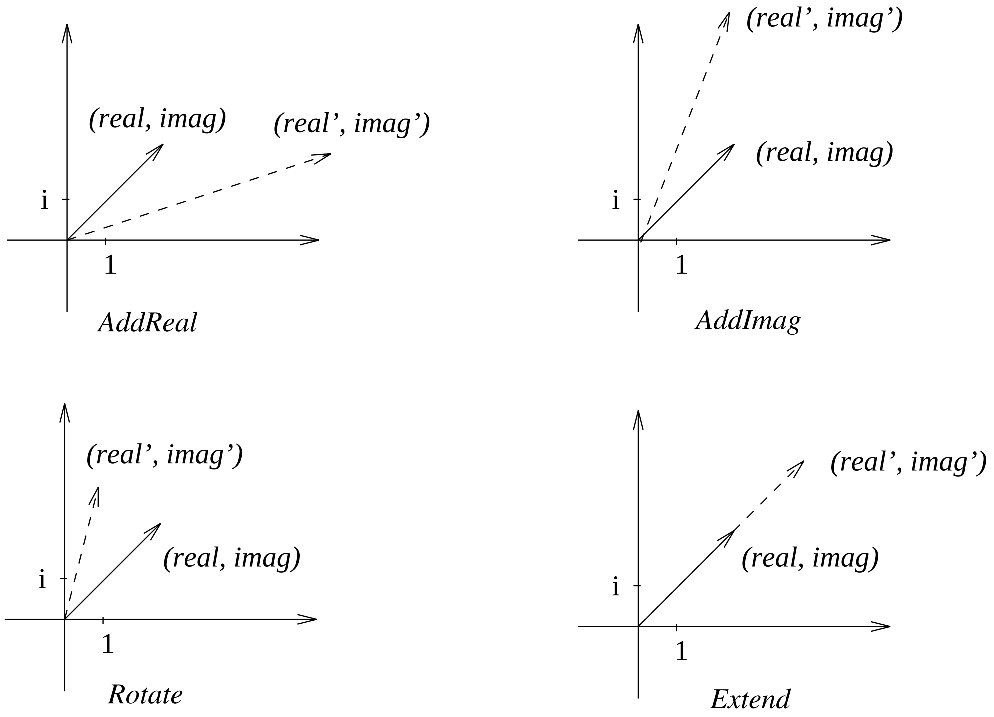


Figure 3.2: Operations to manipulate a complex number variable.

(1) add a given real number to its real part,

(2) add a given real number to its imaginary part,

(3) rotate by a given angle and

(4) extend the modulus by a given positive real number.

When modelling a complex variable as an object, if only two attributes, *real* and *imag*, are considered, then operations *Rotate* and *Extend* (particularly *Rotate*) are difficult and cumbersome to construct. However, if additional secondary attributes, such as the modulus and the argument, are introduced, then operations *Rotate* and *Extend* can be easily defined. A suitable model of a complex variable in Object-Z is the class *Complex*.

Complex

$real, imag : \mathbb{R}$

Δ

$modulus, argument : \mathbb{R}$

$modulus = \sqrt{real^2 + imag^2}$

$modulus * sin(argument) = imag$

$modulus * cos(argument) = real$

AddReal

$\Delta(real)$

$r? : \mathbb{R}$

$real' = real + r?$

AddImag

$\Delta(imag)$

$i? : \mathbb{R}$

$imag' = imag + i?$

Rotate

$\Delta(real, imag)$

$\alpha? : \mathbb{R}$

$argument' = argument + \alpha?$

$modulus' = modulus$

Extend

$\Delta(real, imag)$

$l? : \mathbb{R}^+$

$modulus \neq 0$

$modulus' = modulus + l?$

$argument' = argument$

The dependency of the secondary attributes $(modulus, argument)$ on primary attributes $(real, imag)$ is specified in the class invariant.

The benefit of introducing the secondary attributes *modulus* and *argument* in *Complex* is that operations *Rotate* and *Extend* are easily defined by expressing change in terms of them; moreover, in this case the required changes to the primary attributes *real* and *imag* are unambiguously deducible. Another interesting point demonstrated by this example is that, in a particular situation, the value of a secondary attribute need not be uniquely determined. For instance, when $modulus = 0$, the value of the secondary attribute *argument* is arbitrary and even if $modulus \neq 0$, *argument* is only determined modulo 2π.

3.3 Dependency on Environment: Sharing

In this section, we consider a system in which a secondary attribute plays an important role in capturing a notion of object sharing; also, the value of the secondary attribute is not determined by its objects' primary attributes but by the environment.

3.3.1 A Game Sharing Group

Consider a situation in which a group of personal computer (PC) users share their computer games through a communication device. Each member (player) of the group owns a PC and a exclusive set of computer games stored in the PC. Individual players can add a new game to or delete an existing game from their PC. Any game of any player can be accessed (played) by all players of the group. Games can also be transferred between players. See Figure 3.3 for an illustration. The following is an Object-Z specification of this system.

Firstly, let skeletal class *Game* represent a computer game.

Game

...

Play

...

A player of the group can be modelled as:

Player

$my\_games : \mathbb{P}\, Game$

Δ

$other\_games : \mathbb{P}\, Game$

AddGame

$\Delta(my\_games)$

$g? : Game$

$g? \notin my\_games$

$my\_games' = my\_games \cup \{g?\}$

DelGame

$\Delta(my\_games)$

$g? : Game$

$g? \in my\_games$

$my\_games' = my\_games \setminus \{g?\}$

Choose

$g? : Game$

$g? \in (my\_games \cup other\_games)$

$Play \mathrel{\widehat=} Choose \bullet g?.Play$

The primary attribute *my_games* denotes the local games. The secondary attribute *other_games* denotes all other public games, i.e. games not local to a player but which can be accessed by the player. However such an intention cannot be expressed as an invariant of the class *Player* because, for any object $p : Player$, the value of $p.other\_games$ depends on the games which are owned by players other than p in the group. Therefore the value of $p.other\_games$ is dependent on the environment of p. The precise meaning of this attribute is defined by the state invariant of the system class below.

GameSharingGroup

$$
\begin{array}{l}
players : \mathbb{P}\, Player \\
\Delta \\
games : \mathbb{P}\, Game \\
\hline
\forall\, p_1, p_2 : players \bullet \\
\qquad p_1 \neq p_2 \Rightarrow p_1.my\_games \cap p_2.my\_games = \varnothing \\
games = \bigcup\{p : players \mid p.my\_games\} \\
\forall\, p : players \bullet p.other\_games = games \setminus p.my\_games
\end{array}
$$

Select

$$
\begin{array}{l}
p? : Player \\
\hline
p? \in players
\end{array}
$$

SelectTwo

$$
\begin{array}{l}
p_1?, p_2? : Player \\
\hline
p_1? \neq p_2? \\
\{p_1?, p_2?\} \subseteq players
\end{array}
$$

$$Play \mathrel{\widehat=} Select \bullet p?.Play$$

$$NewGame \mathrel{\widehat=} Select \bullet p?.AddGame$$

$$DiscardGame \mathrel{\widehat=} Select \bullet p?.DelGame$$

$$Transfer \mathrel{\widehat=} SelectTwo \bullet p_1?.DelGame \wedge p_2?.AddGame$$

The system consists of a set of players. The first predicate of the class invariant ensures that each player's *my_games* is a distinct set of games. The other predicates of the class invariant ensures that the value of *other_games* of a player in a group depends on those games which are locally contained by any other player of the group. Incidentally, the definition is facilitated by the introduction of the secondary attribute *games*. This example demonstrates that, given an object $p : Player$ in the group, the value of $p.other\_games$ is determined by the existence of some game objects in the environment and those game objects are not even referred to by the primary attribute $p.my\_games$. Therefore the value $p.other\_games$ is subject to change even without applying an operation to p. To illustrate this in detail, let's consider an instance of *GameSharingGroup* (Figure 3.3).

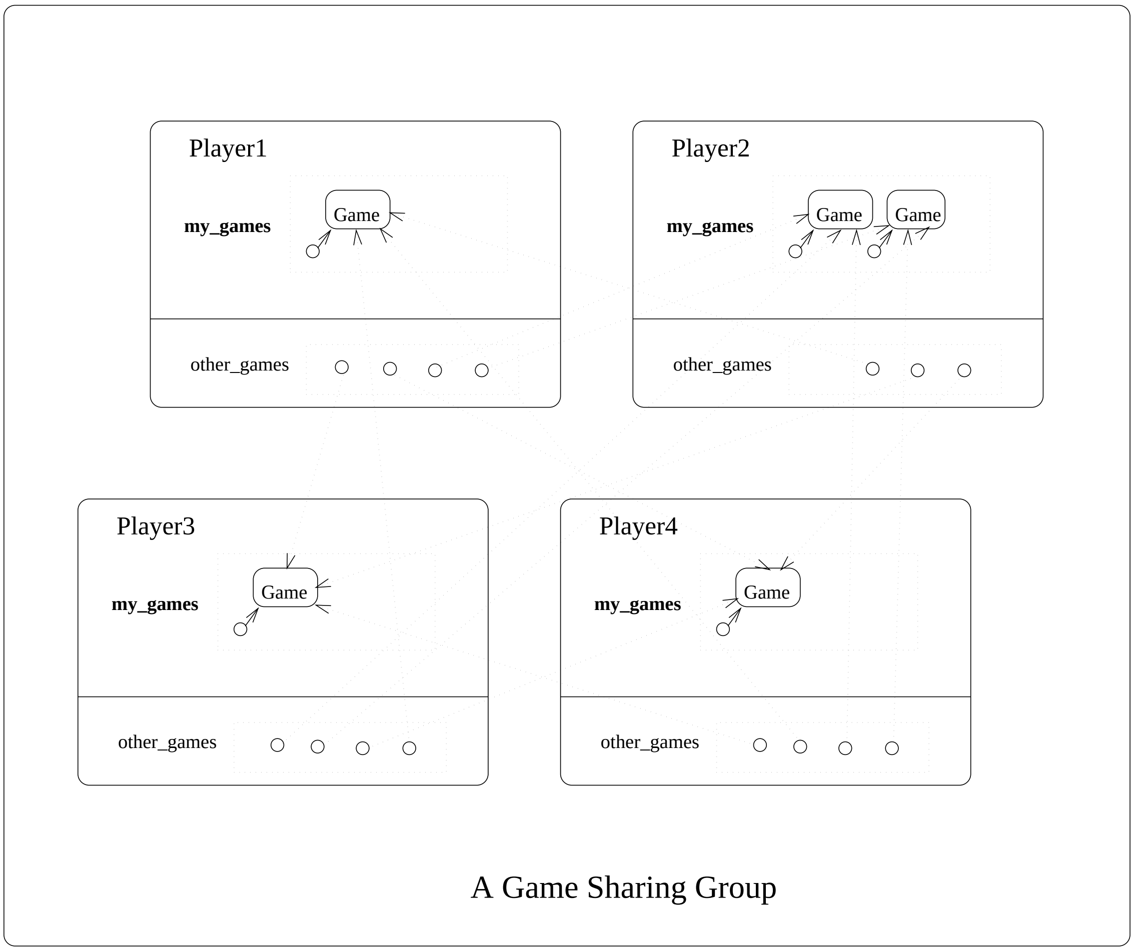


Figure 3.3: A group of PC users sharing games.

Suppose the system performs an operation *DiscardGame* (*Select* $\bullet$ $p?.DelGame$) or *NewGame* (*Select* $\bullet$ $p?.AddGame$); if $Player2$ is selected (i.e. $p? = Player2$) and a game is deleted from (added to) $Player2.my\_games$, then the game is deleted from (added to) all $Player1.other\_games$, $Player3.other\_games$ and $Player4.other\_games$.

Although every game is accessible by all the players of the group, the existence of local games (in my_games) is controlled solely by the player; i.e. for any player p, $p.my\_games$ can be changed only by performing $p.AddGame$ or $p.DiscardGame$ whereas p has no control of the existence of $p.other\_games$. In the definition of *Player*, this difference is captured precisely by distinguishing my_games and $other\_games$ as primary and secondary respectively.

In the next section, we investigate the role of secondary attributes in modelling recursive structures.

3.4 Dependency Chains and Recursion

If a referenced object has the same type as its client object, then the client object is recursively defined. Frequently, a property of a client object depends on its referenced objects. Such dependencies require that the object reference chain from the client object through its referenced objects be acyclic. Secondary attributes can be used to capture both the dependency chain and the acyclic object reference structures in recursive definitions. This situation arises frequently when specifying denotational semantics, and the use of secondary attributes helps to clarify and simplify the specification of the semantic domains.

3.4.1 A Simple Predicates Store

To facilitate this discussion, consider a store of simple predicates. A simple predicate can be a logical variable or a conjunction of a simple predicate or a negation of a simple predicate. The value of a simple predicate can be output from the store and the value of a logical variable can be changed. Notice that the value of a logical variable is the value stored in the variable while the value of a conjunction depends on its left and right components. If the value of a logical variable is changed then the values of all predicates involved with the logical variable are possibly affected.

All predicates in the store are within the same scope (referencing the same set of variables). Figure 3.4 illustrates the syntactic structure of two predicates '`(A and B) and C`' and '`not (B and C)`' which both refer to the same logical variables '`B`' and '`C`'. To model these predicates in Object-Z, three classes model the three kinds of predicates, namely logical variables, conjunctions and negations. The three classes have a common property — a logical value. This common part is modelled as an abstract class *BasePredicate* which is inherited in the definition of the three predicate classes —

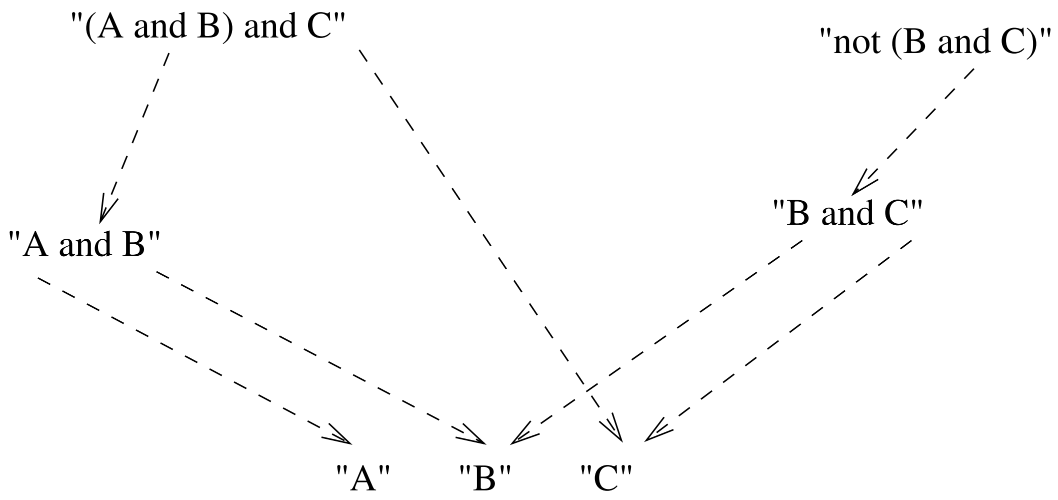


Figure 3.4: Syntax Structures of Two Predicates.

LogicVar, *Conjunction* and *Negation*. (Note that an abstract class is created only for inheritance reuse and it does not have any instance in the system.)

BasePredicate

Δ

$value : \mathbb{B}$

$subs : \mathbb{P}(\downarrow BasePredicate)$

The secondary attribute *value* represents the value of a predicate. Secondary attribute *subs* represents the collection of direct and indirect component object identities: it will facilitate the specification of an acyclic reference structure of any predicate.

A logical variable (an object of class *LogicVar*) has a *name* and a *content*, a boolean value which can be changed by operation *Change*. *LogicVar* is derived by inheritance from *BasePredicate*.

LogicVar

- *BasePredicate*
- State:
  - $name : Id$
  - $content : \mathbb{B}$
  - ---
  - $value = content$
  - $subs = \varnothing$
- *Change*:
  - $\Delta(content)$
  - $value? : \mathbb{B}$
  - ---
  - $content' = value?$

The class invariant '$value = content$' indicates that the value of a variable is that stored in the variable, while '$subs = \varnothing$' indicates that a logical variable has no sub-component objects.

A conjunction predicate consists of a left and a right hand predicate and a negation predicate consists of the predicate which is negated. They are also derived by inheriting *BasePredicate*.

Conjunction

- *BasePredicate*
- State:
  - $l, r : \downarrow BasePredicate$
  - ---
  - $l \notin BasePredicate$
  - $r \notin BasePredicate$
  - $value = (l.value \wedge r.value)$
  - $subs = \{l, r\} \cup l.subs \cup r.subs$
  - $self \notin subs$

Negation

- *BasePredicate*
- State:
  - $p : \downarrow BasePredicate$
  - ---
  - $p \notin BasePredicate$
  - $value = \neg\, p.value$
  - $subs = \{p\} \cup p.subs$
  - $self \notin subs$

Note that the term *self* denotes the identity of the object itself (see [44] for a detailed discussion).

In the definition of *Conjunction* and *Negation*, the class invariants '$l \notin BasePredicate \wedge r \notin BasePredicate$' and '$p \notin BasePredicate$' declare that variables l, r, p are members of the *proper* subclasses of *BasePredicate* not the members of *BasePredicate* itself.

The class invariants '$value = (l.value \wedge r.value)$' and '$value = \neg\ p.value$' ensure that the value of a conjunction or a negation depends on the values of their components and ultimately on the primary attributes (*content*) of the variables. The class invariants involved with the secondary attributes *subs* capture the acyclic property of the object reference structure. (Note that the term *self*[44] denotes the identity of the object itself.) As an illustration, Figure 3.5 shows the state and the object reference structures of the two predicates '`(A and B) and C`' and '`not (B and C)`' of Figure 3.4. The dependency chain from the *value*s of '`(A and B) and C`' and of '`not (B and C)`' to the *value*s of the variables '`A`', '`B`' and '`C`' can be clearly seen; for instance, if the *content* of the variable '`C`' is changed from '`false`' to '`true`', the value of the predicates '`(A and B) and C`' and '`not (B and C)`' will be negated through the object reference structure.

A store of predicates can be modelled as

PredStore

$$preds : \mathbb{P}(\downarrow BasePredicate)$$
$$\Delta$$
$$vars : \mathbb{P}\ LogicVar$$

$$preds \cap BasePredicate = \varnothing$$
$$vars = (\bigcup\{p : preds \bullet p.subs\} \cup preds) \cap LogicVar$$
$$\#\{v : vars \bullet v.name\} = \#vars$$

Output

$$pred? : Predicate$$
$$value! : \mathbb{B}$$

$$pred? \in preds \wedge value! = pred?.value$$

$$Change \mathrel{\widehat=} [var? : vars] \bullet var?.Change$$

The secondary attribute *vars* gathers all occurrences (direct and indirect) of variables in the store. The class invariant $\#\{v : vars \bullet v.name\} = \#vars$ ensures that all

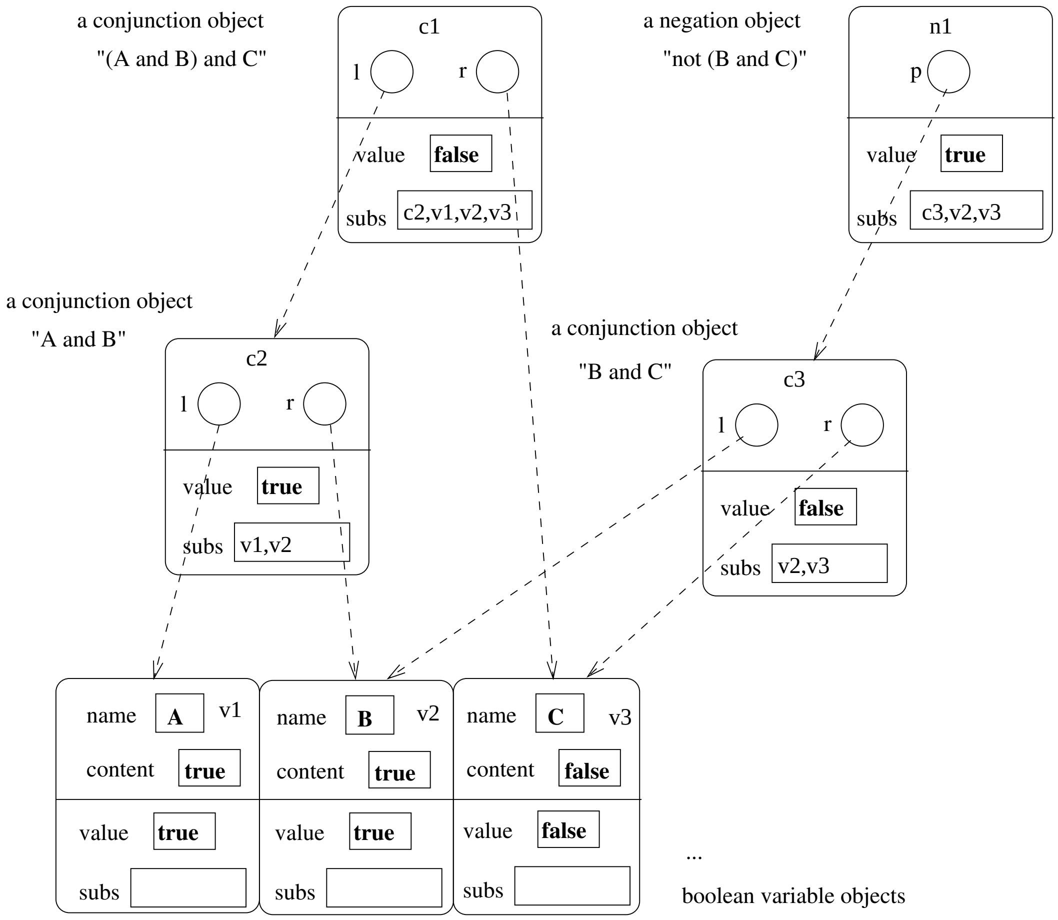


Figure 3.5: Object Structures of the Two Predicates.

variables in the store have distinct names so that all predicates in the store are within the same scope. For instance, in Figure 3.5, '`(A and B) and C`' and '`not (B and C)`' refer to the same variables '`B`' and '`C`'.

The operation *Output* demonstrates that a predicate can be selected from the store and that its value can be broadcast to the environment of the store. The operation *Change* demonstrates that a boolean variable can be selected and its value changed.

In this section, the importance of using secondary attributes to model recursive structures has been clearly demonstrated.

3.5 Conclusion

In this chapter, the implications of applying secondary attributes in formal specification have been explored. Three small case studies analyse and demonstrate three different usages of secondary attributes in formal modelling, namely:

- improving the clarity and simplicity of object-oriented system specifications in general;
- capturing a notion of object sharing; and
- constructing recursive definitions.

We have seen in this chapter how the notion of secondary attributes enables invariant relationships between objects to be clearly specified. This is particularly valuable when designing large or complex computer systems.

The game sharing group case study in Section 3.3 demonstrates that the notions of information distribution and information sharing are facilitated by the use of secondary attributes. It is believed that secondary attributes will contribute significantly to the formal modelling of distributed systems. The ideas for constructing recursive definitions in Section 3.4 will be applied to specify the denotational semantics of programming languages in the following chapters (Chapter 4, 9 and 10). Secondary attributes also facilitate the development of the notations for object containment (in Chapter 7) and for exclusive object control (in Chapter 8) in Object-Z specifications.

Chapter 4

An OO Approach to the Semantics of Programming Languages

The state-based specification languages VDM and Z can be used to define the semantics of programming languages[7, 41, 58, 71, 105]. The approaches of VDM and Z are very similar (see [58] for a comparison of the two languages when applied to the specification of the semantics of a small Pascal-like procedural language).

Usually, the abstract syntax, static semantics and dynamic semantics of a 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. The general lack of modularity and reusability of the traditional denotational semantics representations has been realised in the literature as a drawback[89].

In this chapter, a novel object-oriented approach is taken 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 is used to define the semantics of a very simple procedural programming language.

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 4.1 the semantics of expressions is specified in both Z and Object-Z and the resulting formal structures compared.

Section 4.2 is an object-oriented specification of the syntax and semantics of the small procedural language.

Section 4.3 identifies and discusses some issues that arise in this object-oriented specification of the programming language, and these discussions motivate the development of a generalised polymorphic construct, class-union, and the development of notations for capturing object containment in Object-Z. These new extensions will be presented in the following few chapters.

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

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

4.1 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.

4.1.1 Concrete Syntax of Expressions

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

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

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

4.1.2 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. An abstract syntax of expressions in Z is

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

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:

$$Scaltype ::= booltype \mid inttype$$

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

$$
\begin{array}{|l}
wf_s boolexp, wf_s intexp, wf_s exp : Exp \rightarrow \mathbb{B} \\
typeexp : Exp \nrightarrow Scaltype \\
\hline
\forall\, b : \mathbb{B};\; n : \mathbb{Z};\; e, e_1, e_2 : Exp \bullet \\
\quad wf_s exp(boolconstexp(b)) \\
\quad wf_s exp(intconstexp(n)) \\
\quad wf_s exp(plusexp(e_1, e_2)) \Leftrightarrow wf_s intexp(e_1) \wedge wf_s intexp(e_2) \\
\quad wf_s exp(lessexp(e_1, e_2)) \Leftrightarrow wf_s intexp(e_1) \wedge wf_s intexp(e_2) \\
\quad wf_s exp(andexp(e_1, e_2)) \Leftrightarrow wf_s boolexp(e_1) \wedge wf_s boolexp(e_2) \\
\quad wf_s boolexp(e) \Leftrightarrow wf_s exp(e) \wedge typeexp(e) = booltype \\
\quad wf_s intexp(e) \Leftrightarrow wf_s exp(e) \wedge typeexp(e) = inttype \\
\quad \mathrm{dom}\, typeexp = \{e : Exp \mid wf_s exp(e)\} \\
\quad typeexp(intconstexp(n)) = typeexp(plusexp(e_1, e_2)) = inttype \\
\quad typeexp(boolconstexp(b)) = booltype \\
\quad typeexp(lessexp(e_1, e_2)) = typeexp(andexp(e_1, e_2)) = booltype
\end{array}
$$

The value of an expression can be either boolean or integer. 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:

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

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[1]. However in our object-oriented approach, these global environments are implicitly captured by the semantics domain of Object-Z.

4.1.3 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. A constant can be viewed as an object. A binary expression can also be viewed as an object which contains two 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 or an and-expression according to the concrete syntax of expressions in Section 4.1.

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

[1]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.

$$type\_of : Scalval \rightarrow Scaltype$$

$$\forall\, b : \mathbb{B};\ n : \mathbb{Z} \bullet$$
$$type\_of(boolval(b)) = booltype$$
$$type\_of(intval(n)) = inttype$$

The common property of various expressions can be modelled as an abstract class, and then the abstract class can be inherited to define other classes — constant class and binary expression class.

BaseExp

$$type : Scaltype$$
$$\Delta$$
$$val : Scalval$$
$$exps : \mathbb{P} \downarrow BaseExp$$

$$type\_of(val) = type$$

OutVal

$$val! : Scalval$$

$$val! = val$$

The type of an expression is represented by attribute *type*, and the meaning (value) by attribute *val*. The secondary attribute *exps* represents a collection of all expression object identities contained within an expression and it will be used to capture the acyclic object reference structure of any expression. The operation *OutVal* indicates that the value of an expression can be broadcast.

A constant is modelled as an object of the class *Const* which is defined by inheriting *BaseExp*:

Const

BaseExp

$$content : Scalval$$

$$val = content$$
$$exps = \{self\}$$

The content of a constant is a scalar value which cannot be changed; therefore, there is no operation defined in the class *Const* to change the value of *content*.

The class *BinaryExp* represents the common part of binary expressions.

BinaryExp

BaseExp

$$e_1, e_2 : \downarrow BinaryExp$$

$$e_1 \notin BinaryExp \land e_2 \notin BinaryExp$$
$$self \notin e_1.exps \cup e_2.exps$$
$$exps = e_1.exps \cup e_2.exps \cup \{self\}$$

The class invariant $e_1 \notin BinaryExp \land e_2 \notin BinaryExp$ ensures that variables e_1, e_2 are members of the *proper* subclasses of *BinaryExp* not the members of *BinaryExp* itself.

The class invariants which include the term '*exps*' specify that the object reference structure of an expression is acyclic (e.g. an expression cannot be a sub-expression directly or indirectly of itself).

Classes *PlusExp*, *LessExp* and *AndExp* of specific binary expression objects are defined by inheriting the generic class *BinaryExp*. The meaning (value) of a specific expression object is dependent upon the meaning of its components.

PlusExp

BinaryExp

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

LessExp

BinaryExp

$type = booltype$

$e_1.type = e_2.type = inttype$

$val = boolval(intval^\sim(e_1.val) < intval^\sim(e_2.val))$

AndExp

BinaryExp

$type = booltype$

$e_1.type = e_2.type = booltype$

$val = boolval(boolval^\sim(e_1.val) \wedge boolval^\sim(e_2.val))$

In each expression class the attribute *val* depends in a functional way upon the other attributes.

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 add the following classes:

OrExp

BinaryExp

$type = booltype$

$e_1.type = e_2.type = booltype$

$val = boolval(boolval^\sim(e_1.val) \vee boolval^\sim(e_2.val))$

MultExp

BinaryExp

$type = inttype$

$e_1.type = e_2.type = inttype$

$val = intval(intval^{\sim}(e_1.val) * intval^{\sim}(e_2.val))$

Other classes remain unchanged.

In Section 4.2 we shall show that variable references, too, can be easily included in expressions.

4.2 Language Semantics in Object-Z

4.2.1 Concrete Syntax

The concrete syntax of the small procedural language is:

Program	→	'Program' *Id*
		'dec' *Declist Stmt*
Id	→	*Letter* | *Letter Id*
Letter	→	'a'|'b'...
Declist	→	ϵ | *Declaration Declist*
Declaration	→	*Var* ':' *Type* ';'
Var	→	*Id*
Type	→	'integer' | 'boolean'
Stmt	→	*AssignStmt* | *IfStmt* | *SeqStmt*
AssignStmt	→	*Var* ':=' *Exp* ';'
IfStmt	→	'if' *Exp* 'then' *Stmt* 'else' *Stmt* ';'
Exp	→	...(as in Section 4.1.1) | *Var*
SeqStmt	→	'begin' *Stmtseq* 'end'
Stmtseq	→	ϵ | *Stmt Stmtseq*

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

4.2.2 Expressions

The definition of expressions in Section 4.1.3 is extended by defining a variable reference class *Varef*. *Varef* requires the definition of the class *VarLoc* which represents variables locations.

Varef contains an identifier (*id*) and a reference (*vl*) to a *VarLoc* object. Let

$$[Id]$$

denote the set of all possible identifiers.

VarLoc

$$type : Scaltype$$
$$content : Scalval$$

$$type\_of(content) = type$$

INIT

$$type = booltype \Rightarrow content = boolval(false)$$
$$type = inttype \Rightarrow content = intval(0)$$

Assign

$$\Delta(content)$$
$$val? : Scalval$$

$$content' = val?$$

Varef

BaseExp

$id : Id$
$vl : VarLoc$

$type = vl.type$
$val = vl.content$
$exps = \{self\}$

$Assign \mathrel{\widehat=} vl.Assign$

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

The definitions of *PlusExp*, *LessExp* and *AndExp* remain as in Section 4.1.3.

4.2.3 Statements

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

The common part of statements can be modelled as an abstract class, and then this abstract class can be inherited to define other classes — the *AssignStmt* class, the *IfStmt* class and the *SeqStmt* class.

BaseStmt

Δ
$varrefs : \mathbb{P}\, Varef$
$stmts : \mathbb{P}(\downarrow BaseStmt)$

The secondary attribute *varrefs* is the set of variable references appearing in a statement. The totality of these variable references is used in the predicate part of the

Program class (in Section 4.3.5) to ensure every variable appearing in a program statement also appears in a declaration. The secondary attribute *stmts* represents a collection of all statement object identities contained within a statement and it will be used, as the same purpose as that in defining expressions, to capture the acyclic object reference structure of any statement.

The meaning of an assignment statement, unlike an expression, is not a single value but rather a store transformation. In the semantics below, such transformations are effected by message (value) passing from the right hand side (expression) to the left hand side (variable reference) of an assignment.

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

AssignStmt

$BaseStmt$

$$lhs : Varef$$
$$rhs : \downarrow BaseExp$$

$$rhs \notin BaseExp \wedge rhs \notin BinaryExp$$
$$lhs.type = rhs.type$$
$$varrefs = \{lhs\} \cup (rhs.exps \cap Varef)$$
$$stmts = \{self\}$$

$$Execute \mathrel{\widehat=} rhs.OutVal \parallel lhs.Assign$$

The class invariant $rhs \notin BaseExp \wedge rhs \notin BinaryExp$ indicates that *rhs* refers to a real expression which is not a member of the abstract classes *BaseExp* and *BinaryExp*.

The dependent attribute *varrefs* is the set of variable references appearing in an assignment statement.

Notice that in the definition of the class *AssignStmt*, the abstract syntax of an assignment statement is represented by state attributes *lhs* (left hand side of the assignment — a variable reference) and *rhs* (right hand side of the assignment — an expression). The static semantics of the assignment statement is captured by the state invariant $lhs.type = rhs.type$. The dynamic semantics of the assignment statement is represented by the operation *Execute* which assigns the value of *rhs* to the variable referenced *lhs*. Note that the meaning of a statement, unlike an expression, is not a single value but rather a store transformation. For example, the meaning of an assignment in VDM or Z [58, 71] is represented by overriding the old store to become a new store. In the Object-Z representation above, such a transformation is represented by message passing.

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

IfStmt

$BaseStmt$

$$\begin{array}{l}
test : Exp \\
then, else : \downarrow BaseStmt \\
\hline
then \notin BaseStmt \land else \notin BaseStmt \\
test.type = booltype \\
varrefs = (test.exps \cap Varef) \cup then.varrefs \cup else.varrefs \\
self \notin then.stmts \cup else.stmts \\
stmts = then.stmts \cup else.stmts \cup \{else, then\}
\end{array}$$

$$\begin{array}{ll}
Execute \mathrel{\widehat=} & [boolval^{\sim}(test.val)] \land then.Execute \\
& \quad [] \\
& [\neg\ boolval^{\sim}(test.val)] \land else.Execute
\end{array}$$

The meaning of an *if* statement is represented by the *Execute* operation of the class *IfStmt*. 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 sequence of statements consists of a list of statements. Its meaning is to execute the statements in the sequence one by one. It is modelled as:

SeqStmt

BaseStmt

$stmts : \mathrm{seq}(\downarrow BaseStmt)$

$\mathrm{ran}\ stmts \cap BaseStmt = \varnothing$

$varrefs = \bigcup\{s : \mathrm{ran}\ stmts \bullet s.varrefs\}$

$stmts = \bigcup\{s : \mathrm{ran}\ stmts \bullet s.stmts\} \cup \{self\}$

$self \notin stmts$

$Execute \mathrel{\widehat=} \fatsemi\ i : 1 \mathinner{..} \#stmts \bullet stmt(i).Execute$

Note that the operation *Execute*, namely,

$$\fatsemi\ i : 1 \mathinner{..} \#stmts \bullet stmt(i).Execute$$

is an abbreviation for

$$stmts(1).Execute \fatsemi stmts(2).Execute \fatsemi \ldots \fatsemi stmts(\#stmts).Execute.$$

4.2.4 The Program

A program contains variable declarations, *vdecs*, and a body statement, *stmt*. A program is modelled as an object of the class *Program*. This completes the language semantics.

Program

$vdecs : Id \rightarrowtail\!\!\!\!\rightarrow VarLoc$

$body : Stmt$

$\{vr : body.varrefs \bullet vr.id \mapsto vr.vl\} \subseteq vdecs$

INIT

$\forall\ vl : \mathrm{ran}\ vdecs \bullet vl.\mathrm{INIT}$

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

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

4.3 Discussion

In this chapter an object-oriented approach is applied to the specification of the semantics of a programming language. 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.

4.3.1 Polymorphic Declarations

By revisiting the specification of the simple programming language, it is found that the polymorphic type declaration in Object-Z needs to be further developed. A new type construct is needed to exclude the top abstract class, say 'C', from the polymorphic inheritance hierarchy type '$\downarrow C$'. For instance, in this chapter, the following predicates were explicitly included in the specification of the programming language.

$$e_1 \notin BinaryExp \wedge e_2 \notin BinaryExp \qquad \text{[class invariant of } BinaryExp\text{]}$$
$$rhs \notin BaseExp \wedge rhs \notin BinaryExp \qquad \text{[class invariant of } AssigStmt\text{]}$$
$$then \notin BaseStmt \wedge else \notin BaseStmt \qquad \text{[class invariant of } IfStmt\text{]}$$

These repetitious predicates of the specification indicate that $\downarrow BinaryExp$, $\downarrow BaseExp$ and $\downarrow BaseStmt$ are not the most appropriate polymorphic type declarations for attributes e_1, e_2 (of $BinaryExp$), rhs (of $AssigStmt$) and $then, else$ (of $IfStmt$) respectively. This suggests that a way to declare polymorphic type without using an inheritance hierarchy may be useful. In Chapter 5, a general view of polymorphism is taken and the notion of the class-union construct in Object-Z is introduced.

4.3.2 Common Object Reference Structures

Capturing in the specification the acyclic component object reference structures of expressions and statements involved repetitious predicates. For instance, the predicates:

$$self \notin e_1.exps \cup e_2.exps \qquad \text{[class invariant of } BinaryExp\text{]}$$
$$exps = e_1.exps \cup e_2.exps \cup \{self\}$$

$$self \notin then.stmts \cup else.stmts \qquad \text{[class invariant of } IfStmt\text{]}$$
$$stmts = then.stmts \cup else.stmts \cup \{else, then\}$$

were explicitly included in the specification to capture the acyclic object reference structure of a binary expression or an *if* statement. This suggests that there may be a general way to capture the acyclic object reference structure in Object-Z. In Chapter 7, the notion of object containment is investigated and the Object-Z notation is extended so that common object reference structures can be directly captured within system specifications.

These new extensions to Object-Z, namely class-union and object containment, not only play important roles in the object-oriented definition of programming languages but also prove to be useful for modelling object-oriented systems in general. Therefore the next two chapters present these extensions with the aim of modelling object-oriented systems in general rather than being concerned only with the programming language model. Then in Chapters 9 and 10, these new extensions (class-union and object containment) are applied to present the specifications of a block structured programming language and an object-oriented programming language.

Chapter 5

Polymorphic Class Types

In most commercial object-oriented programming languages such as Eiffel[85, 86] and C++[108], object polymorphism (i.e. dynamic binding, as distinct from genericity) is defined within the subclass structure induced by inheritance. The reasons for this are partly historical and partly pragmatic: the mechanism of inheritance allows for code sharing, and this is a good starting point to capture other requirements such as signature compatibility[112] needed for polymorphism. Languages such as Eiffel go further, taking a strict view of polymorphism and imposing a limited form of behavioural compatibility (subtyping) on inheritance hierarchies. As has long been recognised however [16, 22, 112], the marriage of polymorphism and inheritance can result in conflict when the role of inheritance as simple code sharing is contrasted with the requirements for behavioural compatibility.

This thesis is concerned with specifying (rather than implementing) object-oriented systems and a very general view of polymorphism is adopted. In Section 5.1 we describe informally some simple systems that strongly suggest a form of polymorphism that does not easily fit within an inheritance hierarchy. Indeed, these examples suggest that polymorphism can involve classes not related by subtyping.

In Section 5.2 a class-union construct is defined. This enables the construction of suitable sets of object references where the underlying objects so referenced are not necessarily all of the same class. This construction is applied to the systems informally discussed in Section 5.1, with class-union enabling the polymorphic declaration of object references. Indeed, as explained in Section 5.3, the traditional view of polymorphism expressed within an inheritance hierarchy can also be described using the class-union construct. Finally, Section 5.4 presents a more complex case study of the application of class-union.

As implementational aspects of polymorphism are not of concern here, throughout this chapter polymorphism will be discussed within the context of system specification.

5.1 Polymorphic References

An object reference within (the specification of) a system is said to be *polymorphic* if the class of the object so referenced is not uniquely determined. Polymorphic object references traditionally form the basis for polymorphism in object-oriented systems, both in programming [85] and in formal specification [42]. In this section, however, three examples are considered where polymorphic object references are applied in a non-traditional context.

5.1.1 Example: A Power Tool

A power tool can be fitted with either an attachment for drilling or an attachment for sawing (see Figure 5.1), where such attachments are objects of classes *Drill* or *Saw* specified by

Drill

$DrillState ::= still \mid drilling$

$mode : DrillState$

DrillHole

$\Delta(mode)$

$mode' = drilling$

Stop

$\Delta(mode)$

$mode' = still$

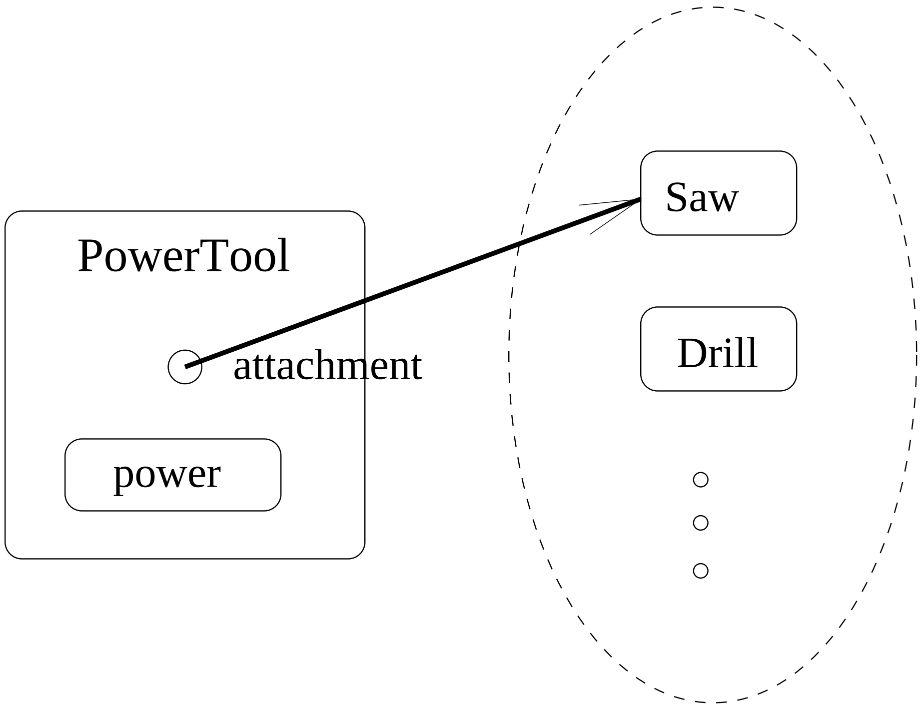


Figure 5.1: A power tool

Saw

$SawState ::= still \mid sawing$

$mode : SawState$

Cut

$\Delta(mode)$

$mode' = sawing$

Stop

$\Delta(mode)$

$mode' = still$

In this case the power tool specification would (informally) look something like

$$
\begin{array}{l}
\textbf{PowerTool} \\
\quad \begin{array}{|l}
att : Drill \ \mathbf{or}\ Saw \\
power\_on : \mathbb{B} \\
\hline
\neg\ power\_on \Rightarrow att.mode = still
\end{array}
\qquad
\begin{array}{|l}
\textit{INIT} \\
\hline
\neg\ power\_on
\end{array} \\
\\
\quad \begin{array}{|l}
SwitchOn \\
\hline
\Delta(power\_on) \\
\hline
\neg\ power\_on \\
power\_on'
\end{array}
\qquad
\begin{array}{|l}
SwitchOff \\
\hline
\Delta(power\_on) \\
\hline
power\_on \\
\neg\ power\_on'
\end{array} \\
\\
\quad \begin{array}{|l}
Change \\
\hline
\Delta(att) \\
att? : Drill \ \mathbf{or}\ Saw \\
\hline
\neg\ power\_on \\
att' = att?
\end{array} \\
\\
\quad DrillHole \mathrel{\widehat=} [att \in Drill] \land SwitchOn \land att.DrillHole \\
\quad Cut \mathrel{\widehat=} [att \in Saw] \land SwitchOn \land att.Cut \\
\quad Stop \mathrel{\widehat=} SwitchOff \land att.Stop
\end{array}
$$

Without, at this stage, precisely defining the semantics of the type *Drill* **or** *Saw*, the intention is that the attribute *att* is a polymorphic object reference: its value is a reference to an attached object in either the *Drill* or *Saw* class. The traditional way to express this polymorphism would be to construct a class *Attachment*, say, of which both *Drill* and *Saw* would be subclasses, and then declare

$$att : \downarrow Attachment.$$

The construction of such a super class, however, is somewhat artificial as distinct attachments may have little in common (except that they must interface with the power tool) and may display completely unrelated functionality.

Notice that the operations *DrillHole* and *Cut* are conditional upon the attribute *att* being in the appropriate class. These operations are reminiscent of the type-down-cast construction in C++. The operation *Stop*, on the other hand, is a polymorphic operation since it is defined irrespective of the class of *att*. Similarly, the class invariant $\neg\ power\_on \Rightarrow att.mode = still$ is interpreted polymorphically.

The *Change* operation enables the attachment to be changed provided the power is turned off. The input to this operation, *att*?, is again a polymorphic object reference.

Figure 5.2: Ginger Meggs

5.1.2 Example: Ginger Meggs

Ginger Meggs[1] (see Figure 5.2) is a boy whose pocket is often crammed with various bits and pieces such as marbles, string, etc. This treasure is constantly being collected

[1]Ginger Meggs is an Australian cartoon character.

and discarded. Objects can be examined, e.g. marbles can be polished, string can be knotted, and so on. An (informal) specification of Ginger Meggs might look something like:

GingerMeggs

$pocket : \mathbb{P}(Marble \textbf{ or } String)$

Collect

$\Delta(pocket)$
$x? : Marble \textbf{ or } String$

$x? \notin pocket$
$pocket' = pocket \cup \{x?\}$

Discard

$\Delta(pocket)$
$x! : Marble \textbf{ or } String$

$x! \in pocket$
$pocket' = pocket - \{x!\}$

SelectMarble

$m? : Marble$

$m? \in pocket$

SelectString

$s? : String$

$s? \in pocket$

$PolishMarble \mathrel{\widehat=} SelectMarble \bullet m?.Polish$

$KnotString \mathrel{\widehat=} SelectString \bullet s?.Knot$

The attribute *pocket* is a set of polymorphic object references: each element of *pocket* is a reference to an object in either the *Marble* or *String* class. The operations *Collect* and *Discard* are specified polymorphically, but the operations *PolishMarble* and *KnotString* can only be applied to objects of the appropriate class.

As specified, Ginger Meggs can only have marbles and string in his pocket. To enable worms and other treasures to be collected the type of *pocket* would need to be extended. Hence any notation for declaring polymorphic object references would need to be easily extendible. Restricting polymorphism to subclass hierarchies has this advantage, as the extendibility of the hierarchy is facilitated by the mechanism of inheritance.

In the case of Ginger Meggs, however, there seems to be no a-priori class structure that would include classes as distinct as *Marble* and *String* and to which classes like *Worm*, say could be added by inheritance. Certainly any such hierarchy would not be considered polymorphic in the traditional sense. Furthermore, if the type of *pocket* is extended it may well be necessary to extend the operations in the class *GingerMeggs* as well, e.g. specific operations for worms would be needed (although what such operations might be is left to the imagination of the reader). However, the operations *Collect* and *Discard* should remain essentially unaltered despite any extension.

5.1.3 Example: A Channel

One kind of polymorphism supported by to some degree all programming languages is function and operator overloading. For example, in procedural languages the assignment statement is a polymorphic operator because it supports both the assignment of the value of a boolean expression to a boolean variable, and the assignment of the value of an integer expression to an integer variable. The following example is based on this idea.

Classes *Nat* and *Bool* are defined as follows:

Nat

$n : \mathbb{N}$

INIT
$n = 0$

Update
$\Delta(n)$
$x? : \mathbb{N}$

$n' = n + x?$

OutVal
$x! : \mathbb{N}$

$x! = n$

Bool

$b : \mathbb{B}$

INIT
$b = true$

Update
$\Delta(b)$
$x? : \mathbb{B}$

$b' = x?$

OutVal
$x! : \mathbb{B}$

$x! = b$

The behaviour of objects of the classes *Nat* and *Bool* are completely different: the interface and functionality of even the common-named operations *Update* and *OutVal* are different. However, consider now the class *Channel* specified as follows:

$$
\begin{array}{|l}
\underline{\textit{Channel}} \\
\begin{array}{|l}
\hline
sender, receiver : Nat \textbf{ or } Bool \\
\hline
sender \neq receiver \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\underline{\textit{Switch}} \\
\Delta(sender, receiver) \\
s?, r? : Nat \textbf{ or } Bool \\
\hline
sender' = s? \land receiver' = r? \\
\hline
\end{array} \\
\\
Trans \mathrel{\widehat{=}} [\{sender, receiver\} \subseteq Nat] \bullet sender.OutVal \parallel receiver.Update \\
\qquad\qquad [] \\
\qquad\qquad [\{sender, receiver\} \subseteq Bool] \bullet sender.OutVal \parallel receiver.Update \\
\hline
\end{array}
$$

The attributes *sender* and *receiver* are polymorphic object references. Another kind of polymorphism is represented by the *Trans* operation: this operation can be performed on the objects *sender* and *receiver* (providing they are of the same kind) regardless of whether they are both in class *Nat* or class *Bool*. However, the component operations *OutVal* and *Update* are not polymorphic because the type of the parameters is different. The *Trans* operation essentially sets up a polymorphic channel that enables two objects of *Nat* or two objects of *Bool* to communicate.

5.2 Class Union

Underlying the semantics of an Object-Z specification is the understanding that each object has a unique persistent identity that distinguishes it from every other object, regardless of class[44]. An object's identity is invariant irrespective of changes to its state.

As one semantic role of a class denotes the set of identities of possible objects of the class, then in any system being specified, let set $\mathbb{O}$ denote the universe of all object identities, regardless of class, that may be in the system.[2]

If A and B are distinct classes in the system,

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

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 $\mathbb{O}$. In this case, if we define $C \mathrel{\widehat{=}} A \cup B$ then C is not itself a class (see Figure 5.3) but only a set of object identities.

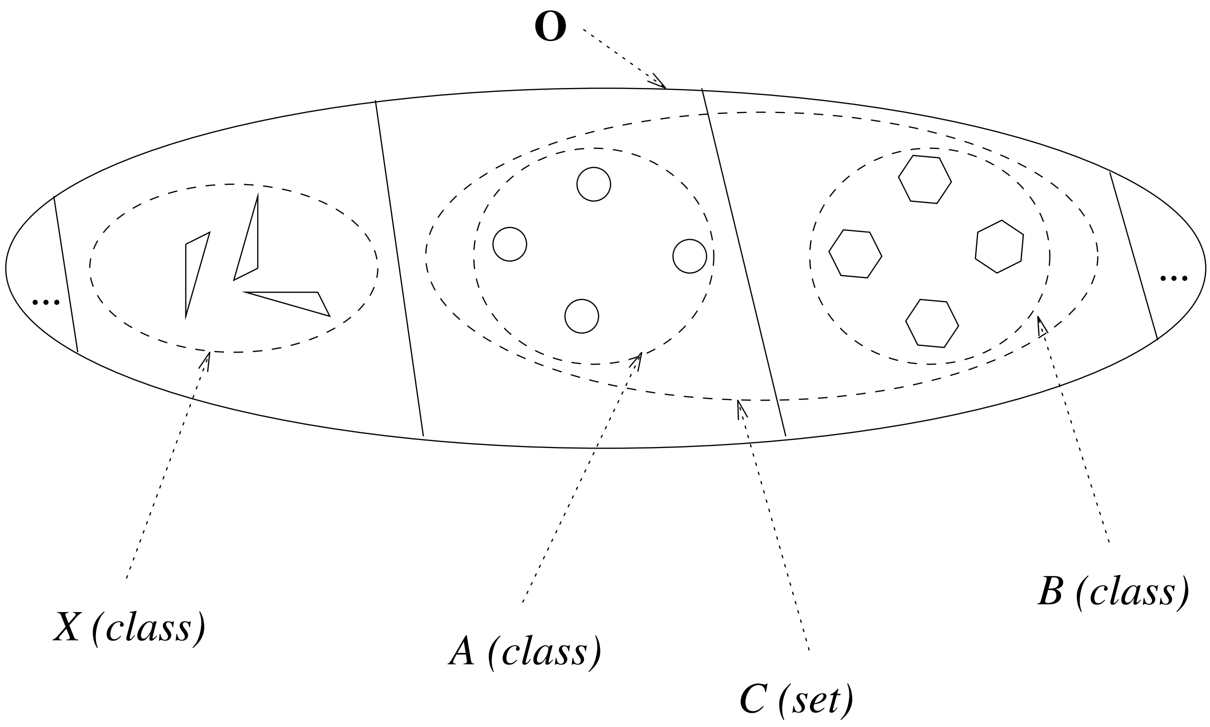


Figure 5.3: Class Union

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

5.2.1 Polymorphic Core

Associated with any object is a set of attributes and a set of operations. Given a collection of objects, possibly from different classes, an attribute named *attr* of type T, say, is said to be *polymorphic with respect to the collection* if every object in the collection has an associated attribute named *attr* of type T. 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 parameter list of the operation Op is the same, i.e. the input and output parameters are identically named and typed.

The notion of polymorphic operation is similar to signature compatibility[3][112] and is introduced for the same reason: it establishes a minimal condition under which messages to objects declared polymorphically can be interpreted regardless of the actual class to which they belong.

There are differences, however, between this notion of polymorphic operation and the traditional view of signature compatibility in object-oriented systems. We do not insist that, for instance, an operation common to the objects in the collection must be polymorphic; collections will be permitted where common-named operations have distinct parameter lists. Furthermore, there need not be a distinguished object, or set of objects, in the collection whose attributes and operations are each associated polymorphically with all the other objects in the collection; i.e. we shall not suppose the existence of a parent class effectively defining by inheritance the set of polymorphic attributes and operations.

Associated with any collection of objects is the set of all attributes and operations that are polymorphic with respect to the collection. This set of attributes and operations will be called the *polymorphic core* of the collection. For instance, if

$$C \mathrel{\widehat{=}} A \cup B$$

[3]Signature compatibility requires that, if two objects have the same named operation, the interface of that operation must be the same in both cases.

where A and B are the following classes

A

$$a, c, x : \mathbb{Z}$$

Change

$$\Delta(a)$$
$$i? : \mathbb{Z}$$
$$a' = x + i?$$

ChangeC

$$\Delta(c)$$
$$i? : \mathbb{Z}$$
$$c' = x + i?$$

B

$$b, d, x : \mathbb{Z}$$

Change

$$\Delta(b)$$
$$i? : \mathbb{Z}$$
$$b' = x * i?$$

ChangeD

$$\Delta(d)$$
$$i? : \mathbb{Z}$$
$$d' = x * i?$$

then the polymorphic core of C is the attribute x and the operation *Change*.

5.2.2 Typing

Given a declaration $s : C$, the expressions '$s.x$' and '$s.Change$' can be interpreted polymorphically regardless of the actual class of s; consequently, the expressions '$s.x$' and '$s.Change$' are well typed under the type scope '$s : C$'. In contrast, the expressions '$s.a$' and '$s.ChangeC$' conflict with the situation when the actual class of s is B because 'a' and '$ChangeC$' are features in A but not in B. Therefore, those expressions '$s.a$' and '$s.ChangeC$' are considered as ill-typed (under the type scope '$s : C$') unless the type of s is narrowed ('$\succ\!\!\div$') to the subtype — class A, i.e. the expressions '$s \succ\!\!\div A \bullet s.a$' and '$[s \succ\!\!\div A] \bullet s.ChangeC$' are well typed.

In general, suppose a reference e is originally declared as '$e : E$' (where E is a class-union), then the expressions '$e \succ\!\!\div F \bullet pred(e.attr)$' and '$[e \succ\!\!\div F] \bullet e.op$' are well typed (or valid) if

- F is a class in E or is a class-union of classes which are included in E, i.e. $F \subseteq E$, and
- the attribute *attr* and the operation *op* are in the polymorphic core of F.

The expressions '$e \succ\!\!\div F \bullet pred(e.attr)$' and '$[e \succ\!\!\div F] \bullet e.op$' are logically equivalent to '$\forall\, e_0 : F \bullet e_0 = e \Rightarrow pred(e_0.attr)$ ' and '$[e_0 : F \mid e_0 = e] \bullet e_0.op$' respectively.

Notice that if $F \subseteq E$, then the polymorphic core of F is (often) larger than or (at least) equal to the polymorphic core of E. Type narrowing facilitates the type checking for polymorphic references in Object-Z. It is reminiscent of downcasting in C++.

5.2.3 Examples Revisited

With the notions of class union and polymorphic operations, the Section 5.1 examples can be revisited and the specification of polymorphic object references formalised.

Power Tool

In the class *PowerTool* the attribute *att* can now be declared as

$$att : Attachment$$

where

$$Attachment \mathrel{\widehat=} Drill \cup Saw.$$

The operation *Stop* is in the polymorphic core of the collection *Attachment* and consequently this operation can be used polymorphically in the specification. The other operations, *DrillHole* and *Cut*, must be qualified by a type narrowing rather than a predicate involving the specific type (the subtype) of the attachment:

$$DrillHole \mathrel{\widehat=} [att \succ\!\!\div Drill] \bullet (SwitchOn \land att.DrillHole)$$
$$Cut \mathrel{\widehat=} [att \succ\!\!\div Saw] \bullet (SwitchOn \land att.Cut)$$

The operation *Change* is an operation to manipulate the attachments, rather than to operate upon the attachments themselves; the input $att?$ can now also be declared as

$$att? : Attachment.$$

Ginger Meggs

In the class *GingerMeggs* the attribute *pocket* can now be declared as

$$pocket : \mathbb{P}\ Treasure$$

where

$$Treasure \mathrel{\widehat{=}} Marble \cup String.$$

The polymorphic core of *Treasure* is empty so all operations upon the contents of the pocket must be qualified by a type restriction involving object type. Similarly, the parameters $x?$ and $x!$ in the operations *Collect* and *Discard* will also have the type *Treasure*.

Communication Channel

In the class *Channel* the attributes *sender* and *receiver* can now be declared as

$$sender, receiver : Objects$$

where

$$Objects \mathrel{\widehat{=}} Nat \cup Bool.$$

The inputs $s?$, $r?$ in the operation *Switch* will also have the type *Objects*. The polymorphic core of *Objects* is empty despite the fact that the operations *Update* and *OutVal* occur in both *Nat* and *Bool*; the parameter list of these operations differ between the classes. The operation *Trans* needs to be modified to be statically type checked:

$$\begin{aligned} Trans \mathrel{\widehat{=}}\ & [sender, receiver \succ\!\!\div Nat] \bullet sender.OutVal \parallel receiver.Update \\ & [] \\ & [sender, receiver \succ\!\!\div Bool] \bullet sender.OutVal \parallel receiver.Update \end{aligned}$$

5.2.4 Extending Class Unions

The specification of the power tool in Sections 5.1.1 and 5.2.2 has the property that the value of the attribute *att* is restricted to being a reference to an object in either the *Drill* or *Saw* class. To extend the specification of the power tool to also accept a plane attachment, i.e. an object of class *Plane*, say, it would be necessary to redefine

$$Attachment \mathrel{\widehat{=}} Drill \cup Saw \cup Plane.$$

The existing specification of the class *PowerTool* remains valid under such an extension of the class union type provided the *Plane* class has an operation *Stop* compatible with the same-named operation in the *Drill* and *Saw* classes. If such an operation does exist in the *Plane* class, the polymorphic application of *Stop* in the *PowerTool* class remains valid.

This observation suggests the following rule:

> a class-union type can be extended, without affecting existing specifications using that type, provided the extended type has the same polymorphic core as the original.

That is, extension of class-union types is permitted if the polymorphic core is preserved[4].

In the specification of Ginger Meggs, the polymorphic core of the class-union *Treasure* is empty. Hence the type *Treasure* can be extended to include class *Worm*, etc. without affecting the existing specification.

5.3 Class Union and Traditional Polymorphism

The class union construction facilitates the declaration of polymorphic object references. But the traditional notation $\downarrow C$ where C is a class also facilitates such declarations. In this section we compare the two points of view which we see as complementary.

[4]In specific cases, extension may be possible even if the polymorphic core is not preserved, provided those operations not preserved are not applied polymorphically.

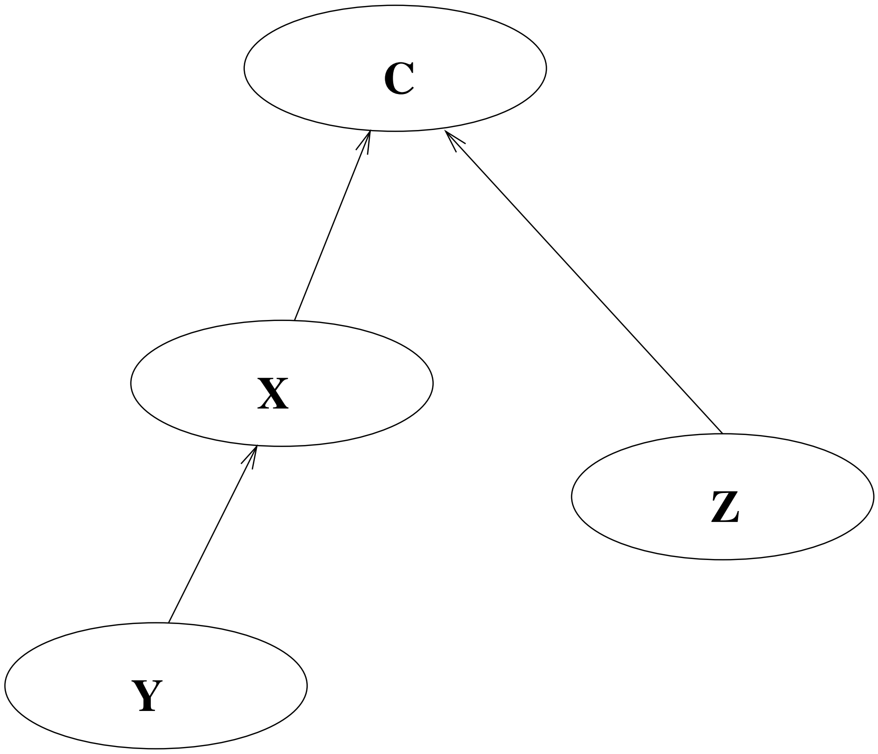


Figure 5.4: An Inheritance Hierarchy

Consider the inheritance hierarchy in Figure 5.4 (the actual features of the classes will not be of importance). A declaration $c : \downarrow C$ indicates that the value of c is a polymorphic reference to an object in one of the classes C, X, Y or Z. Such an intention could also have been captured by the declaration

$$c : C \cup X \cup Y \cup Z.$$

However, there are significant distinctions between the way in which these declarations are applied within a specification. For instance, the notation $\downarrow C$ is used only if the class hierarchy with C at its root satisfies strict signature compatibility[112], namely, all the operations defined in C must be inherited in all the classes below C in the hierarchy, and in each class an operation inherited from C must have an identical parameter list. In other words, the polymorphic core of the union of all the classes in the hierarchy determined by C is precisely the set of operations defined in C. This rule is implicit in the notation $\downarrow C$ but not in the class-union notation which is designed to be applicable in wider contexts.

Given an inheritance hierarchy, say in Figure 5.4, using class-union it is also possible to declare

$$d : C \cup X \cup Y$$

where class Z is deliberately excluded. Such a selection may be made, for example, when only part of the inheritance hierarchy satisfies strict signature compatibility: those parts not conforming to the requirements of operation polymorphism can be excluded. Hence it is possible to take inheritance hierarchies that do not conform to the strict requirements for polymorphism throughout, but can never-the-less be used polymorphically after pruning using the class-union construct. The selection of such sub-structures using the notation $\downarrow C$ is not possible.

5.4 Case Study: A Telephone System

In this section, class union is applied to the specification of a simple telephone system. This system consists of a collection of telephones that each conform to the core operations of sending and receiving calls, etc. but which can exhibit quite different functionality outside this core. The class-union construct is ideal for specifying such systems.

We take a greatly simplified view of a telephone system, concentrating upon the issue of inter-phone connection. A more detailed Object-Z specification capturing other aspects of a telephone system is given in [78].

5.4.1 Informal View of the System

The telephone system specified here consists of a collection of one-line and multi-line telephones, and considers the connections that can be made between them.

A one-line telephone can be connected to, at most, one other telephone at any one time. When busy, such a telephone cannot send or receive additional calls, i.e. all attempts to dial a busy one-line telephone will fail.

A multi-line telephone, on the other hand, can engage in several calls at once. Even when busy, incoming calls can be accepted and placed on hold. Likewise, outgoing calls can be made after first placing all existing connections on hold.

As a simplification, we shall ignore issues such as ringing telephones only being connected after the receiver is lifted: we shall suppose that connection occurs immediately when a call is successfully sent.

5.4.2 Specification of the System

At any time a one-line telephone is either idle or busy, while a multi-line telephone is either idle, holding or busy. Hence define

$$
\begin{aligned}
OnePhState &::= idle \mid busy \\
MultiPhState &::= idle \mid holding \mid busy.
\end{aligned}
$$

A one-line telephone is an object of the class *OneLinePh*, while a multi-line telephone is an object of the class *MultiLinePh*. In the specification of both classes it will be necessary to declare polymorphic object references whose value is a reference to objects of either class. Hence define the class union

$$Phone \mathrel{\widehat{=}} OneLinePh \cup MultLinePh.$$

OneLinePh

$$
\begin{array}{|l}
s : OnePhState \\
current : Phone \\
\hline
s = idle \Rightarrow current = self
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textsc{Init}} \\
\hline
s = idle
\end{array}
$$

Send

$$\begin{array}{l} \Delta(s, current) \\ to!, from! : Phone \\ \hline from! = self \\ to! \neq self \\ s = idle \\ s' = busy \\ current' = to! \end{array}$$

Receive

$$\begin{array}{l} \Delta(s, current) \\ to?, from? : Phone \\ \hline to? = self \\ from? \neq self \\ s = idle \\ s' = busy \\ current' = from? \end{array}$$

Hangup

$$\begin{array}{l} \Delta(s, current) \\ to!, from! : Phone \\ \hline s = busy \\ s' = idle \\ to! = current \\ from! = self \end{array}$$

Cutoff

$$\begin{array}{l} \Delta(s, current) \\ to?, from? : Phone \\ \hline s = busy \\ s' = idle \\ to? = self \\ from? = current \end{array}$$

The attribute *current* denotes the connected telephone, if there is one. The input/output parameters of the operations *Send*, *Receive*, *Hangup* and *Cutoff* are object references, so that communication within a telephone system is modelled by the passing of object identities rather than the passing of detailed state information.

MultLinePh

$$
\begin{array}{l}
max : \mathbb{N} \\
lines : \mathbb{P}\, Phone \\
s : MultiPhState \\
current : Phone \\
\hline
\# lines \leq max \\
lines = \varnothing \Rightarrow s = idle \\
s \in \{idle, holding\} \Rightarrow current = self
\end{array}
$$

INIT

$$
lines = \varnothing
$$

Send

$$
\begin{array}{l}
\Delta(s, lines, current) \\
to!, from! : Phone \\
\hline
from! = self \\
to! \notin lines \\
to! \neq self \\
\# lines < max \\
s \in \{idle, holding\} \\
s' = busy \\
lines' = lines \cup \{to!\} \\
current' = to!
\end{array}
$$

Receive

$$
\begin{array}{l}
\Delta(s, lines) \\
to?, from? : Phone \\
\hline
to? = self \\
from? \notin lines \\
from? \neq self \\
\# lines < max \\
s \in \{idle, holding\} \Rightarrow s' = holding \\
s = busy \Rightarrow s' = busy \\
lines' = lines \cup \{from?\}
\end{array}
$$

Retrieve

$$
\begin{array}{l}
\Delta(s, current) \\
\hline
s = holding \\
s' = busy \\
\exists\, p : lines \bullet current' = p
\end{array}
$$

PutOnHold

$$
\begin{array}{l}
\Delta(s, current) \\
\hline
s = busy \\
s' = holding
\end{array}
$$

Hangup

$\Delta(s, lines, current)$
$to!, from! : Phone$

$to! = current$
$from! = self$
$s = busy$
$lines' = lines \setminus \{current\}$
$lines' \neq \varnothing \Rightarrow s' = holding$

Cutoff

$\Delta(s, lines, current)$
$to?, from? : Phone$

$s \in \{holding, busy\}$
$to? = self$
$from? \in lines$
$lines' = lines \setminus \{from?\}$
$from? = current \wedge lines' \neq \varnothing \Rightarrow s' = holding$
$from? \neq current \wedge lines' \neq \varnothing \Rightarrow (s' = s \wedge current' = current)$

The attribute *lines* denotes the set of telephones currently connected, where the number of such connections is limited by the value of max. The attribute *current* denotes the connected telephone not on hold, if there is one.

Compared with a one-line telephone, an additional two operations, *Retrieve* and *PutOnHold*, are defined. Furthermore, the functionality of the operations *Send*, *Receive*, *Answer*, *Hangup* and *Cutoff* is enhanced to reflect the fact that a multi-line telephone can be involved in several connections simultaneously.

A telephone system can now be specified as an aggregation of telephones undergoing connections and disconnections.

PhoneSystem

$$ps : \mathbb{P}\, Phone$$
$$pairs : Phone \leftrightarrow Phone$$
$$pairs \subseteq ps \times ps$$

INIT

$$\forall\, p : ps \bullet p.\textsc{Init}$$
$$pairs = \varnothing$$

AddTwo

$$p_1, p_2 : Phone$$
$$\neg\ p_1 \ \underline{pairs}\ p_2$$
$$pairs' = pairs \cup \{(p_1, p_2), (p_2, p_1)\}$$

DelTwo

$$p_1, p_2 : Phone$$
$$p_1 \ \underline{pairs}\ p_2$$
$$pairs' = pairs \setminus \{(p_1, p_2), (p_2, p_1)\}$$

$$Connect \mathrel{\widehat=} AddTwo \bullet p_1.Send \parallel p_2.Receive$$

$$Disconnect \mathrel{\widehat=} DelTwo \bullet p_1.Hangup \parallel p_2.Cutoff$$

$$Retrieve \mathrel{\widehat=} [p : MultLinePh \mid p \in ps] \bullet p.Retrieve$$

$$PutOnHold \mathrel{\widehat=} [p : MultLinePh \mid p \in ps] \bullet p.PutOnHold$$

The state attribute *ps* denotes the set of telephones in the system, while the attribute *pairs* records the (symmetric) connections between the telephones. The operations *Connect* and *Disconnect* establish or terminate the connection between any two telephones, regardless of the type of the telephones, because the operations *Send*, *Receive*, *Hangup* and *Cutoff*, each having the same parameter list in either the class *OneLinePh* or *MultiLinePh*, are all in the polymorphic core of the class union *Phone*. The operations *Retrieve* and *PutOnHold* need to be qualified with a condition that the telephone is multi-line, as these operations are only defined for such telephones.

To complete this section, we consider the problem of specifying the telephone system without using class union. One approach would be to define the class *MultiLinePh* by inheriting the class *OneLinePh* and adding the operations *Retrieve* and *PutOnHold*. The class union *Phone* could then be replaced by $\downarrow$*OneLinePh*. However, the *Send* and *Receive* operations of *MultiLinePh* have distinct pre- and post-conditions, so inheritance would only work if arbitrary redefinition of operations were permitted in polymorphic inheritance hierarchies.

Another approach would be to define *OneLinePh* by inheriting *MultiLinePh* and setting $max = 1$, i.e. consider a one-line telephone to be a multi-line telephone with only one line. Such an inheritance hierarchy is reasonable, but only in hindsight. It would also mean that the operations *Retrieve* and *PutOnHold* would need to be defined for one-line telephones.

A third approach would be to define an abstract *PHONE* superclass with both *OneLinePh* and *MultiLinePh* as subclasses. The skeleton of operations common to the subclasses can be lifted to the superclass, with the detail being left to the subclasses. This illustrates the use of a 'deferred' class (e.g. as in Eiffel) in designing for reusability.

5.5 Conclusion

There are many systems where the application of polymorphism only to suitable inheritance hierarchies is unnecessarily restrictive. In this chapter we considered examples where a polymorphic object reference could be declared whose value referenced an object in one of several classes not related within an inheritance hierarchy. Furthermore, these examples illustrated that even when object references are declared polymorphically, it is often desirable to specify operations that qualify, and hence further restrict, the type of the object in some way.

This thesis' approach to the issues raised by these examples was to define a class-union construct. This construction was then applied, and its implications explored, in several different contexts. Rules for the extension of specifications using class union were formalised which generalised the idea of signature compatibility as usually understood in polymorphic inheritance hierarchies.

The traditional view of object-oriented polymorphism can be seen as a restricted form of class union. As such restrictions are often desirable in practice, the class-union construct is seen as complementary to the marriage of polymorphism with inheritance. Because the class union construct is more widely applicable, however, it enables polymorphism to be applied in situations where the building of an inheritance structure would be cumbersome, or in the case when only part of an inheritance hierarchy satisfies the strict requirements for polymorphism. In Chapters 9 and 10, the class-union construct is used to specify the semantics of programming languages and is compared with the approach in Chapter 4.

The next chapter compares the class-union construct with the free type construct.

Chapter 6

Free Type and Class Union

In the formal specification language Z, the free type construct is introduced as a convention to represent (polymorphic) union types (see Section 2.2 for a brief introduction). Recursive structures can also be modelled by free type definitions. As Object-Z is an object-oriented extension of Z, the free type construct remains a legitimate mechanism to capture polymorphic and recursive structures with Object-Z. On the other hand, the class-union construct (introduced in Chapter 5) is also designed to facilitate the specification of polymorphic and recursive structures. Free type and class-union have been developed from different points of view: the free type construct is based more on the notion of functional values while the class-union construct is based more on the notion of object references. This means that the usages of these two constructs in system modelling are different and it is therefore important to study the differences between these two constructs so that the appropriate one can be chosen when specifying systems. This chapter presents a comparison of the free type and the class-union constructs and provides guidelines for applying them in system modelling.

Section 6.1 presents a case study specifying a currency exchange office which contains bank notes in different currencies. In the case study, the free type and the class-union constructs are applied separately to model the polymorphic type of a bank note, and the resultant representations are compared. Extendibility of the representations is also discussed. Continuing with the discussion on extendibility, Section 6.2 demonstrates that free type and class union can be combined to improve the extendibility of the specification of the communication channel of Section 5.1.3. Section 6.3 presents a case study of a recursive binary tree structure, with an operation to insert a node to a non-empty tree. Using the two constructs to give different specifications of the binary tree further clarifies the distinctions between the functional value point of view (free type) and the object reference point of view (class-union). Suitability of the two constructs for modelling other recursive structures is also discussed. Finally, in Section 6.4, the consistency issue of free type definitions [4, 99] is extended to class-union.

Figure 6.1: Bank Notes.

6.1 Polymorphic Structures

Consider a situation where a currency exchange office (CEO) consists of a collection of franc and dollar bank notes. Suppose an operation can be performed in a CEO to output the total value of the bank notes in dollars (see Figure 6.1). Although the two currencies can be both modelled as the same mathematical kind ($\mathbb{N}$), it is necessary to label each note to distinguish the two different kinds of currency.

The general type of bank note is a polymorphic type which can be modelled as a free type or as a class-union.

Firstly, consider the general type of a bank note specified by a free type.

6.1.1 Modelling Bank Notes in Free Type

If a bank note is modelled as

$$Note_{ft} ::= franc\langle\langle \mathbb{N} \rangle\rangle \mid dollar\langle\langle \mathbb{N} \rangle\rangle$$

(where the sub-script '$_{ft}$' indicates that the term is defined by a free type) then two notes of one currency having the same value, e.g. two ten dollar notes, will be indistinguishable. Therefore bank note identity needs to be modelled in the definition.

Let $[FId]$ and $[DId]$ represent the identities of franc and dollar bank notes (like the serial numbers of the bank notes) respectively. The two currencies can be modelled as

$$\begin{array}{|l} Francs : FId \nrightarrow \mathbb{N} \\ Dollars : DId \nrightarrow \mathbb{N} \end{array}$$

where the ranges of the functions represent the face value of the bank notes. Then, the general type of a bank note can be represented as:

$$BankNote_{ft} ::= franc\langle\langle Francs \rangle\rangle \mid dollar\langle\langle Dollars \rangle\rangle$$

Suppose the exchange rate from francs to dollars is

$$\begin{array}{|l} \mathcal{F}to\mathcal{D} : \mathbb{R} \end{array}$$

(We assume that all arithmetic operators which are defined for $\mathbb{N}$ are also valid for $\mathbb{R}$, and that $\mathbb{N}$ is a subtype of $\mathbb{R}$.)

A CEO can be modelled as

CEO_{ft}

$notes : \mathbb{P}\, BankNote_{ft}$

$TotalInDollars$

$total! : \mathbb{R}$

$$total! = \sum n : (notes \cap \operatorname{ran} dollar) \bullet second(dollar^{\sim}(n)) + \sum n : (notes \cap \operatorname{ran} franc) \bullet (second(franc^{\sim}(n)) * \mathcal{F}to\mathcal{D})$$

(Function *second* denotes the second element of an ordered pair.)

Consider now the use of the class-union construct to model the bank notes.

6.1.2 Modelling Bank Notes in Class Union

If the two kinds of bank notes are modelled as objects of two different classes, *Franc* and *Dollar*, then a general type of a bank note can be defined as a class-union:

$$BankNote \mathrel{\widehat{=}} Franc \cup Dollar$$

where *Franc* and *Dollar* are specified as:

$Franc$

$face\_value : \mathbb{N}$

Δ

$dlr : \mathbb{R}$

$dlr = face\_value * \mathcal{F}to\mathcal{D}$

$Dollar$

$face\_value : \mathbb{N}$

Δ

$dlr : \mathbb{R}$

$dlr = face\_value$

In the classes *Franc* and *Dollar*, the secondary attribute *dlr* (dollar) is used to represent the value of a bank note in dollars. (For a detailed discussion on secondary attributes see Chapter 3.) Now the specification of the CEO can be represented as

$$
\begin{array}{|l}
\textit{CEO} \\
\hline
\begin{array}{|l}
\hline
notes : \mathbb{P}\, BankNote \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\textit{TotalInDollars} \\
\hline
total! : \mathbb{R} \\
\hline
total! = \sum n : notes \bullet n.dlr \\
\hline
\end{array} \\
\hline
\end{array}
$$

Comparing these two models, CEO_{ft} and CEO, we see that one of the differences is that the identity (serial number) of a bank note is implicitly modelled by the semantics of the classes *Franc* and *Dollar* in CEO (a class denotes a set of object identities of the class), while in CEO_{ft}, the serial number of each bank note must be explicitly specified by given types *FId* and *DId*. Another observation is that the definition of the operation *TotalInDollars* in CEO is much simpler than that in CEO_{ft}. This is because the attribute *dlr* is in the polymorphic core of the class-union *BankNote* so that the term '$n.dlr$' in the operation *TotalInDollars* of CEO is interpreted polymorphically regardless of the class of n. In contrast, the operation *TotalInDollars* of CEO_{ft} must be specified using case analysis.

Extendibility

If we extend the specification to include Yen bank notes in the currency exchange office with the exchange rate from yen to dollars as

$$| \; \mathcal{Y}to\mathcal{D} : \mathbb{R},$$

then in the free type representation we need to

- introduce

  $$| \; Yen : YId \twoheadrightarrow \mathbb{N}$$

  where *YId* represents all the serial numbers of Yen notes;

- extend the free type definition $BankNote_{ft}$ to

$$BankNote_{ft} ::= franc\langle\langle Francs\rangle\rangle \mid dollar\langle\langle Dollars\rangle\rangle \mid yen\langle\langle Yen\rangle\rangle;$$

- and modify the predicate part of the operation *TotalInDollars* to become

$$\begin{aligned} total! = {} & \sum n : (notes \cap \operatorname{ran} dollar) \bullet second(dollar^{\sim}(n)) + \\ & \sum n : (notes \cap \operatorname{ran} franc) \bullet (second(franc^{\sim}(n)) * \mathcal{F}to\mathcal{D}) + \\ & \sum n : (notes \cap \operatorname{ran} yen) \bullet (second(yen^{\sim}(n)) * \mathcal{Y}to\mathcal{D}) \end{aligned}$$

However in the class-union representation, we need only

- add the class:

Yen

$$\begin{array}{|l} face\_value : \mathbb{N} \\ \Delta \\ dlr : \mathbb{R} \\ \hline dlr = face\_value * \mathcal{Y}to\mathcal{D} \end{array}$$

- and extend the class-union *BankNote* to

$$BankNote \mathrel{\widehat=} Franc \cup Dollar \cup Yen$$

The specification of the class *CEO* remains unchanged. The class-union representation of a CEO is more extendible because the features in the polymorphic core of a class-union can be applied polymorphically; in contrast, there is no notion of polymorphic core in a free type definition. This example clearly favours the class-union representation. However, this does not suggest the complete replacement of the free type construct by the class-union construct in Object-Z specifications. In the following section, an example demonstrates that a free type can be used to construct the polymorphic core of a class-union in a specification so that the combination of the two improves the extendibility of the specification.

6.2 Construction of a Polymorphic Core

Suppose the channel specification (in Section 5.1.3) is extended to allow a string of characters (in addition to natural numbers and boolean values) to be passed through the channel. The objects which are allowed to be passed through the channel are then represented as:

$$Object \mathrel{\widehat{=}} Nat \cup Bool \cup String$$

(where *Nat* and *Bool* are defined in Section 5.1.3) and the class *String* is defined as

String

$s : \mathrm{seq}\, Char$

Init

$n = \langle\, \rangle$

Update

$\Delta(s)$

$x? : \mathrm{seq}\, Char$

$s' = x?$

OutVal

$x! : \mathrm{seq}\, Char$

$x! = s$

The operation *Trans* of the class *Channel* must also be extended to become

$$\begin{aligned} Trans \mathrel{\widehat{=}} {} & [sender, receiver \succ\!\!\div Nat] \bullet sender.OutVal \parallel receiver.Update \\ & [] \\ & [sender, receiver \succ\!\!\div Bool] \bullet sender.OutVal \parallel receiver.Update \\ & [] \\ & [sender, receiver \succ\!\!\div String] \bullet sender.OutVal \parallel receiver.Update \end{aligned}$$

The representation of the operation *Trans* above is obviously awkward. The reason for the repetitious definition of *Trans* is that the component operations *OutVal* and

Update cannot be applied polymorphically since they are not in the polymorphic core of the class-union *Object* (the parameter types of the operations' input-output interfaces are different). However, by using a free type to represent the general value type of $\mathbb{N}$, $\mathbb{B}$ and seq *Char*, i.e.

$$Val ::= nat\langle\langle \mathbb{N} \rangle\rangle \mid bool\langle\langle \mathbb{B} \rangle\rangle \mid str\langle\langle \mathrm{seq}\ Char \rangle\rangle$$

and re-specifying *Nat*, *Bool* and *String* as

$$
\begin{array}{l}
\hline
\textit{Nat} \\
\hline
n : Val \\
\hline
n \in \mathrm{ran}\ nat \\
\hline
\textsc{Init} \\
n = nat(0) \\
\hline
Update \\
\Delta(n) \\
x? : Val \\
\hline
x? \in \mathrm{ran}\ nat \\
n' = nat(nat^{\sim}(n) + \\
\qquad\qquad nat^{\sim}(x?)) \\
\hline
OutVal \\
x! : Val \\
\hline
x! = n \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\hline
\textit{Bool} \\
\hline
b : Val \\
\hline
b \in \mathrm{ran}\ bool \\
\hline
\textsc{Init} \\
n = bool(true) \\
\hline
Update \\
\Delta(b) \\
x? : Val \\
\hline
x? \in \mathrm{ran}\ bool \\
b' = x? \\
\hline
OutVal \\
x! : Val \\
\hline
x! = b \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\hline
\textit{String} \\
\hline
s : Val \\
\hline
s \in \mathrm{ran}\ str \\
\hline
\textsc{Init} \\
n = str(\langle\,\rangle) \\
\hline
Update \\
\Delta(s) \\
x? : Val \\
\hline
x? \in \mathrm{ran}\ str \\
s' = x? \\
\hline
OutVal \\
x! : Val \\
\hline
x! = s \\
\hline
\end{array}
$$

the operations *Update* and *OutVal* now belong to the polymorphic core of the class-union *Object*. The *Trans* operation in the class *Channel* can now be specified polymorphically as

$$Trans \mathrel{\widehat{=}} sender.OutVal \parallel receiver.Update.$$

Note that the precondition of the operation *Update* in each class *Nat*, *Bool* and *String* ensures that the operation succeeds only if *sender* and *receiver* belong to the same class.

Furthermore, if *Object* is extended to include more new kinds of objects, the operation *Trans* remains unchanged so long as the polymorphic core of the original class-union *Object* is preserved, i.e. the use of free type ensures the specification of the channel is easily extendible.

In the next section, we investigate the role of applying free types and class-unions to model recursive structures.

6.3 Recursive Structures

Consider an ordered binary tree of integers with an operation *Insert* to insert a node into an existing tree. The general structure of the tree consists of nodes that can be modelled using a free type definition

$$TreeNode_{ft} ::= nilnode \mid valnode\langle\langle TreeNode_{ft} \times \mathbb{Z} \times TreeNode_{ft}\rangle\rangle$$

Based on this free type structure, ordered binary tree nodes can be modelled as

$$\begin{array}{|l}
OrderedTreeNode : \mathbb{P}\ TreeNode_{ft} \\
\hline
\forall t : TreeNode_{ft} \bullet t \in OrderedTreeNode \Leftrightarrow \\
\qquad t = nilnode \lor \\
\qquad \exists i : \mathbb{Z};\ l, r : TreeNode_{ft} \bullet \\
\qquad\qquad t = valnode(l, i, r) \\
\qquad\qquad \{l, r\} \subseteq OrderedTreeNode \\
\qquad\qquad (\forall n : nums(l) \bullet n < i) \land (\forall n : nums(r) \bullet n \geq i)
\end{array}$$

where *nums* is an auxiliary function which takes a tree and returns the set of numbers in the tree:

$$
\begin{array}{|l}
nums : TreeNode_{ft} \rightarrow \mathbb{P}\,\mathbb{Z} \\
\hline
\forall\, t : TreeNode_{ft} \bullet \\
\qquad\qquad t = nilnode \Rightarrow nums(t) = \varnothing \\
\qquad\qquad \exists\, i : \mathbb{Z};\ l, r : TreeNode_{ft} \bullet \\
\qquad\qquad\qquad t = valnode(l, i, r) \Rightarrow \\
\qquad\qquad\qquad\qquad nums(t) = nums(l) \cup \{i\} \cup nums(r)
\end{array}
$$

The *Insert* operation can be modelled as a function which takes an integer and a given root node and returns a new tree node:

$$
\begin{array}{|l}
Insert_{ft} : \mathbb{Z} \times OrderedTreeNode \rightarrow OrderedTreeNode \\
\hline
\forall (i, root) : \mathbb{Z} \times OrderedTreeNode \bullet \\
\qquad\qquad root = nilnode \Rightarrow \\
\qquad\qquad\qquad Insert_{ft}(i, root) = valnode(nilnode, i, nilnode) \\
\qquad\qquad \exists\, j : \mathbb{Z};\ l, r : OrderedTreeNode \bullet \\
\qquad\qquad\quad root = valnode(l, j, r) \Rightarrow \\
\qquad\qquad\qquad (i > j \Rightarrow Insert_{ft}(i, root) = valnode(l, j, Insert_{ft}(i, r)) \wedge \\
\qquad\qquad\qquad\ \ i \leq j \Rightarrow Insert_{ft}(i, root) = valnode(Insert_{ft}(i, l), j, r))
\end{array}
$$

Now we consider the representation of the binary tree using the class-union construct. If the binary tree nodes are viewed as objects, they can be modelled as a class-union:

$$TreeNode \mathrel{\widehat{=}} NilNode \cup ValNode$$

where the *NilNode* class represents the *nilnode* objects

$$
\begin{array}{|l|}
\hline
\textit{NilNode} \\
\hline
\begin{array}{|l}
\Delta \\
nodes : \mathbb{P}\ TreeNode \\
\hline
nodes = \{self\}
\end{array} \\
\hline
\end{array}
$$

and the *ValNode* class represents non-nil tree nodes

ValNode

$$
\begin{array}{|l}
val : \mathbb{Z} \\
l, r : TreeNode \\
\Delta \\
nodes : \mathbb{P}\, TreeNode \\
\hline
\langle l.nodes, \{self\}, r.nodes \rangle \text{ partitions } nodes
\end{array}
$$

INIT

$$
\begin{array}{|l}
l \in NilNode \\
r \in NilNode
\end{array}
$$

ToLeft

$$
\begin{array}{|l}
i? : \mathbb{Z} \\
\hline
i? < val
\end{array}
$$

ToRight

$$
\begin{array}{|l}
i? : \mathbb{Z} \\
\hline
i? \geq val
\end{array}
$$

ChangeL

$$
\begin{array}{|l}
\Delta(l) \\
i? : \mathbb{Z} \\
\hline
l \in NilNode \\
l' \in ValNode \\
l'.\textsc{Init} \wedge l'.val = i?
\end{array}
$$

ChangeR

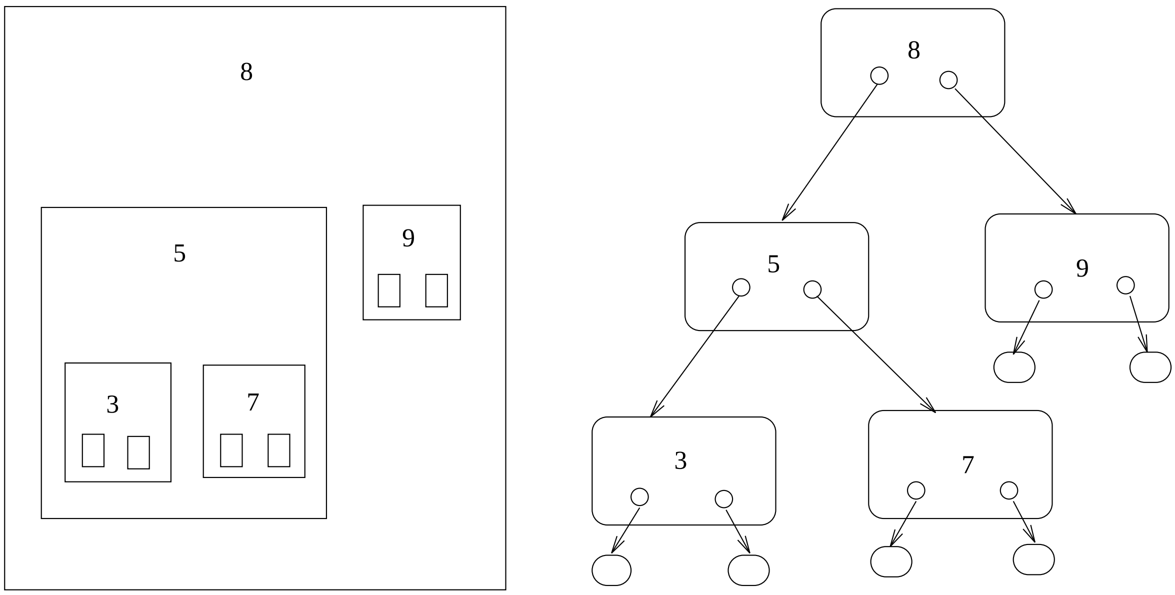
$$
\begin{array}{|l}
\Delta(r) \\
i? : \mathbb{Z} \\
\hline
r \in NilNode \\
r' \in ValNode \\
r'.\textsc{Init} \wedge r'.val = i?
\end{array}
$$

$$
\begin{array}{rl}
Insert \mathrel{\widehat=} & ToLeft \bullet (ChangeL \mathbin{[]} ([l \mathrel{\dot{\asymp}} ValNode] \bullet l.Insert)) \\
& [] \\
& ToRight \bullet (ChangeR \mathbin{[]} ([r \mathrel{\dot{\asymp}} ValNode] \bullet r.Insert))
\end{array}
$$

Secondary attribute *nodes* represents the collection of direct and indirect included tree node object identities. Similar to the secondary attribute *subs* in modelling a logic predicate in Section 3.4, *nodes* plays the role of ensuring the tree object reference structure of a *ValNode* object.

The pre-condition for the *Insert* operation and the state of INIT ensures the order property of a binary tree.

Now consider the following two specifications of a non-empty tree object with an



free type representation

class-union representation

Figure 6.2: Representations of a binary tree.

operation to insert a node at the root of the tree (one is specified by using the free type representation of a tree node and the other is specified by using the class-union):

$Tree_{ft}$

$root : OrderedTreeNode$

INIT

$\exists\, i : \mathbb{Z} \bullet root =$
$\quad valnode(nilnode, i, nilnode)$

Insert

$\Delta(root)$
$i? : \mathbb{Z}$

$root' = Insert_{ft}(i?, root)$

$Tree$

$root : ValNode$

INIT

$root.\mathit{INIT}$

$Insert \mathrel{\widehat=} [i? : \mathbb{Z}] \bullet root.Insert$

In $Tree_{ft}$ (using free type), the value of *root* is changed whenever the operation *Insert* is applied; in contrast, in *Tree* (using class-union), only a particular component tree node of the root tree *root* changes its value when the operation *Insert* is applied; the value of *root* is unchanged because the value of *root* is the identity of the root node of the tree.

The two representations of a binary tree in free type and in class-union are very different in style. This difference is due to the different views (value compared with object reference) which free type and class union are based on. Figure 6.2 captures these different views of a binary tree pictorially.

In this example, it is debatable which representation (by using free type or class union) is more suitable for describing a binary tree. The preference depends on how one views the context.

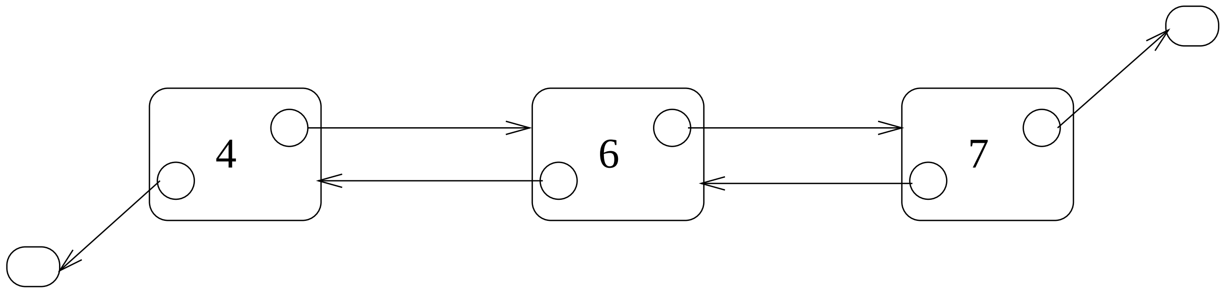


Figure 6.3: A doubly linked list.

However, in general, the class-union representation is more suitable for modelling recursive structures because the reference semantics of an Object-Z class supports cross referencing (object sharing). For instance, a doubly linked list node (of Figure 6.3) can be specified in class-union as

$$DLLNode \mathrel{\widehat=} NilNode \cup DLLValueNode$$

where *DLLValueNode* is specified as:

DLLValueNode

$val : \mathbb{Z}$

$next, prev : DLLNode$

$next \mathrel{\ominus} DLLValueNode \bullet next.prev = self$

$prev \mathrel{\ominus} DLLValueNode \bullet prev.next = self$

INIT

$\{next, prev\} \subseteq NilNode$

[details of operations omitted]

On the other hand, it is impossible to represent the doubly linked list structure using free types because the components of a free type are partitioned; sharing between them is not possible.

6.4 Consistency of Definitions

If it is possible to find a model which satisfies a specification, then the specification is *consistent*. In general, it is undecidable whether or not a Z or Object-Z specification is consistent. However, discussion of this issue can help specifiers avoid writing meaningless specifications. Based on the work on consistency of recursive free types by Arthan[4] and Smith[99], we discuss the consistency issue of the class-union construct.

Consider the following free type definition:

$$X_{ft} ::= set\langle\langle \mathbb{P}\, X_{ft} \rangle\rangle \mid int\langle\langle \mathbb{Z} \rangle\rangle$$

There is no total injective function from $\mathbb{P}\, X_{ft}$ to X_{ft} because, however large X_{ft} is, there are many more members of $\mathbb{P}\, X_{ft}$ than there are members of X_{ft}. Therefore a model for the free type X_{ft} cannot exist. (The detail of the proof can be found in [4, 99].)

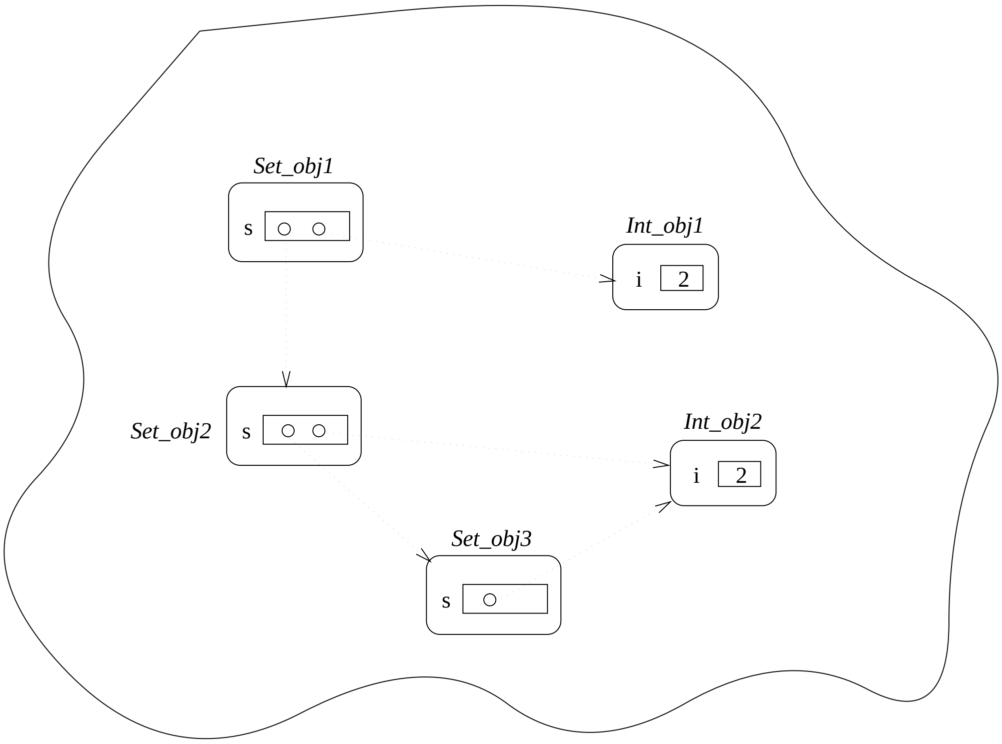


Figure 6.4: A instance model of X

Compare this with the class-union:

$$X \mathrel{\widehat{=}} Set \cup Int$$

where Set and Int are:

Set	Int
$s : \mathbb{P}\, X$	$i : \mathbb{Z}$

As a class-union denotes a set of object identities rather than a set of possible object state values, the value of the attribute s in Set is a subset of object identities in X. Furthermore, self, cross and cyclic object referencing is allowed in class definitions; there are no restrictions on what value s can take. Therefore the representation for X is consistent. For example, Figure 6.4 illustrates a model of X, where $Set = \{Set\_obj1, Set\_obj2, Set\_obj3\}$ and $Int = \{Int\_obj1, Int\_obj2\}$.

As another example consider the free type definition.

$$Y_{ft} ::= fun\langle\langle Y_{ft} \to \mathbb{Z}\rangle\rangle \mid int\langle\langle \mathbb{Z}\rangle\rangle$$

Again, the free type Y_{ft} does not exist as there is no total injective function from $Y_{ft} \to \mathbb{Z}$ to Y_{ft}.

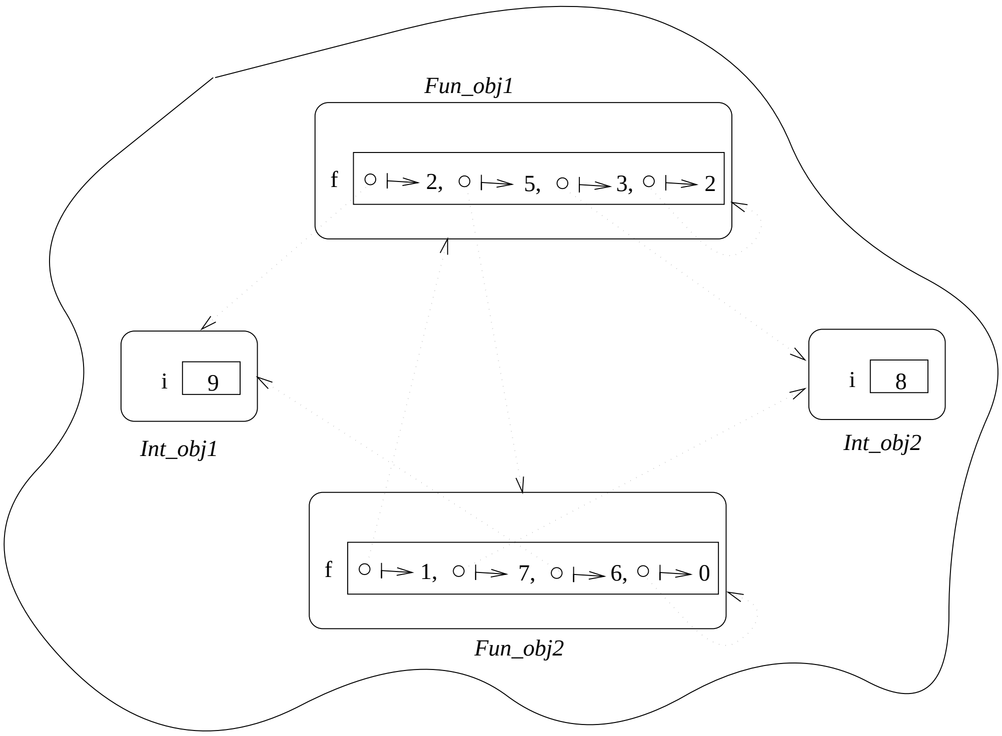


Figure 6.5: A instance model of Y

Compare this with the class-union:

$$Y \mathrel{\widehat{=}} Fun \cup Int$$

where Fun and Int are:

Fun

$f : Y \to \mathbb{Z}$

Int

$i : \mathbb{Z}$

This representation for Y is consistent. For example, Figure 6.5 illustrates a model of Y where $Fun = \{Fun\_obj1, Fun\_obj2\}$ and $Int = \{Int\_obj1, Int\_obj2\}$.

6.5 Conclusion

In this chapter, free type and class-union constructs have been applied to specify a number of polymorphic and recursive systems. By comparing these specifications, our conclusions are:

- The notion of the polymorphic core of a class-union supports the extendibility of system specifications. In contrast, there is no such notion in a free type.
- However, a free type definition can contribute to the construction of the polymorphic core of a class-union so that a combination of the two can improve the extendibility of a system specification. This idea has been successfully applied to specify the denotational semantics of programming languages in Chapters 4 and 9.
- Free type and class-union are based on different views, namely the functional value point of view and the object reference point of view. These different views are particularly evident when recursive structures are modelled. Furthermore, the functional value point of view of free types cannot be applied to modelling recursive structures which involve cross-referencing (sharing).
- The inconsistency cases in free type definitions do not arise in similar class-union structures.

As was illustrated in the binary tree example, it is, in general, debatable which representation, free type or class-union, is best suited for modelling a system. However, if the identity of a polymorphic entity is as important as the content of the entity (e.g. as in the bank note example), then class-union is the better choice. This chapter has provided useful guidelines on how to model polymorphic and recursive structures by choosing the appropriate constructs in Object-Z specifications. These guidelines have influenced our approach to the semantics of programming language models in Chapters 9 and 10.

Chapter 7

Object Containment

In object-oriented systems, references between objects are maintained so as to facilitate inter-object communication[11, 49, 85]. For example, consider a banking system consisting of account objects, customer objects and bank objects. In such a system, a customer object will have attributes whose values reference account objects. These references enable customers to operate their accounts (e.g. to make deposits, withdrawals, etc.). If a bank permits shared accounts, several customers may even reference the same account. In addition, an account object may well have an attribute whose value is a reference to a customer object, so that access to the account can be authorised. Furthermore, a bank object will have attributes whose values reference account objects, so that the bank can operate the accounts for the purpose of adding charges or changing credit limits.

The association between objects determined by the object references in a system will generally result in a complex structure whose design and specification is a crucial part of the development and implementation of the system. An aim of this thesis is to look at ways of capturing formally object reference structures that occur frequently in object-oriented systems.

As an example, suppose that in a banking system no account is shared between banks. In this case it would follow that any account referenced by one bank is distinct from any account referenced by a different bank. When giving a formal specification of this banking system, such an important structural property of the object references should be clearly captured by the specification, ideally as a global system invariant.

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

In this chapter we formally investigate the general nature of the object references implicit in the last two examples. The properties of such object references are captured within a formal framework and incorporated into the Object-Z specification language as predicate rules (resembling in flavour the axioms of combinatorial geometric structures such as projective planes).

The example above of the car and its wheels suggests a notion of geographical location (a car physically contains its wheels) and for this reason we refer to the resulting object-references as *containment*, i.e. a car object is said to (directly) *contain* the wheel objects it references. As the banking system illustrates, however, the structure of object containment also arises in situations where the objects have no relevant geographical location: it would be inappropriate to think of an account object as being physically contained within a bank object, but nevertheless, because distinct banks reference distinct accounts, a bank object is said to (directly) contain the account objects it references. In the thesis this notion of object containment is precisely characterised by its (geometric) properties.

In general, some attributes of an object will reference contained objects, and other attributes will not. For example, in addition to the references to its contained wheel objects, a car object may well reference a person object corresponding to the owner of the car. We would not expect the person reference to denote object containment as a person may well own several cars. That is, in general an object 'has a' set of references to other objects, only some of which may be 'contained' references in the sense defined in this thesis.

Various notions of object association, such as aggregation and composition, have been discussed in the literature[20, 21, 27, 62, 80, 90, 91]. Our notion of containment has features in common with them, but is not identical to any. A notion of containment is also defined by Kilov and Ross[74] in extended Object-Z and used (as a hierarchical subordination) in a way different from the treatment in this thesis.

In this thesis, the properties implied by object containment are modelled by a constraint relationship between objects. Two examples are presented in Section 7.1 to demonstrate that this containment relationship can be captured explicitly in Object-Z by predicates in the state invariant of a class. However, when a system is large and complex, capturing the properties of the containment relationship explicitly in this way is cumbersome. Therefore, in Section 7.2 an extension of the Object-Z notation is introduced to capture directly the geometric notion of object containment.

The notion of object containment is also partially supported by some object-oriented programming languages. Object containment in object-oriented programming languages such as Eiffel[86] (*expanded* class type) and C++[108] (object value type) is discussed and compared to our notion in Section 7.3.

In many object-oriented systems, object containment is closely related to object access: an object can access any object it contains. However, in this thesis we view object containment as a purely geometric notion, quite distinct from the issue of object access. An example in Section 7.4 illustrates this. Chapter 8 further discusses the distinction between the notions of object containment and exclusive object control.

Objects may overlap or share contained objects, e.g. two rooms may share a wall in a building. An object may also contain only part of another object, e.g. a street may pass though and hence be partially contained by several suburbs. In Section 7.5 this generalised view of object containment is illustrated and formally captured in Object-Z.

Finally, the geometry of object containment can be applied to simplify the specification of abstract recursive structures, such as trees and directed acyclic graphs. Examples are presented in Section 7.6.

7.1 Capturing the Properties of Containment

In this section the properties of object containment are stated precisely. The notion of object containment suggests a forest-like geometric relationship between contained objects. As a motivating example, in Figure 7.1 u, v, w, x, y and z denote objects where u directly contains w and x, and similarly w directly contains y and z, but u and v are not related by containment, and neither are w and x. In this case object u *indirectly* contains y and z.

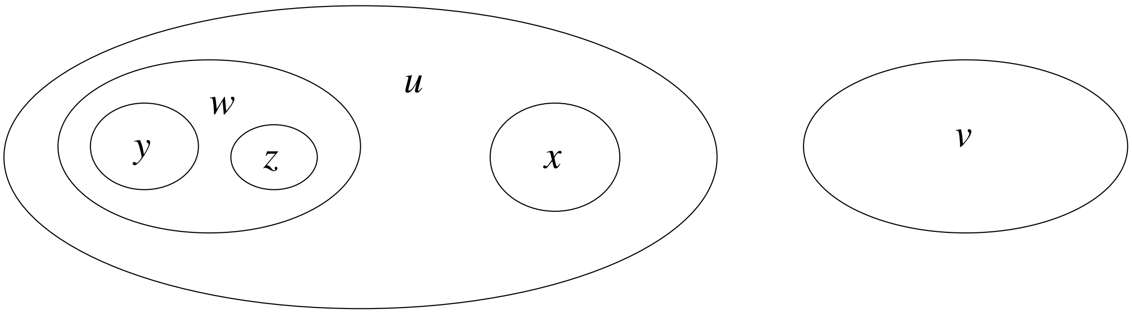


Figure 7.1: The geometry of object containment

Figure 7.1 is based on an idea of geographical object location. However, object containment can arise in systems where the objects are not related geographically. Nevertheless, the figure suggests two geometric ideas that characterise the general notion of object containment[1]:

(1) an object cannot directly or indirectly contain itself; and

(2) an object cannot be directly contained within two distinct objects.

To capture these ideas formally, let

$$dcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

denote the relation whereby

$$ob_1 \; \underline{dcon} \; ob_2$$

if and only if object ob_1 has a reference to a <u>d</u>irectly <u>con</u>tained object ob_2. Put another way, $dcon$ is the set of all those ordered pairs (ob_1, ob_2) of (identities of) objects in the system where ob_1 directly contains ob_2.

[1] In this thesis, the term 'geometry' is used in a combinatorial sense. A geometric structure is defined by the rules that determine whether or not one object 'contains' another. Like projective geometries, no geographical notion of physical location is implied by the term.

The first condition above requires that

$$\nexists ob : \mathbb{O} \bullet ob \ \underline{dcon^{+}} \ ob$$

where $dcon^{+}$ is the transitive closure of $dcon$. The second condition requires that

$$dcon^{\sim} \in \mathbb{O} \nrightarrow \mathbb{O}$$

(i.e. the inverse of the relation $dcon$) is a partial function.

In any system, the relation $dcon$ is not static; it will change dynamically if object relocation is permitted, i.e. if there are operations in the system that affect the containment geometry. This is discussed further in Section 7.2.4.

When specifying an object-oriented system using Object-Z, the above two properties of object containment can be captured explicitly by class invariants, as is illustrated in the following two examples.

7.1.1 Example: Terminal Location

Consider the situation where a campus consists of a set of buildings, with each building containing a set of rooms and each room containing a (possibly empty) set of terminals. A specification in Object-Z would be

Terminal

[details of state and operations omitted]

Room

$ts : \mathbb{P}\ Terminal$

[details of operations omitted]

Building

$$rs : \mathbb{P}\, Room$$

$$\begin{array}{l} \forall\, r_1, r_2 : rs \bullet \\ \quad r_1 \neq r_2 \Rightarrow \\ \qquad r_1.ts \cap r_2.ts = \varnothing \end{array}$$

[details of operations omitted]

Campus

$$bs : \mathbb{P}\, Building$$

$$\begin{array}{l} \forall\, b_1, b_2 : bs \bullet \\ \quad b_1 \neq b_2 \Rightarrow \\ \qquad b_1.rs \cap b_2.rs = \varnothing \\ \qquad \forall\, r_1 : b_1.rs;\ r_2 : b_2.rs \bullet \\ \qquad\qquad r_1.ts \cap r_2.ts = \varnothing \end{array}$$

[details of operations omitted]

The class invariant of the *Building* class captures the idea that no terminal can be in two distinct rooms in a building. Similarly, the class invariant of the *Campus* class captures the idea that no room can be in two distinct buildings of the campus. Furthermore, despite the fact that the predicate of the *Building* class states that no terminal can be in two distinct rooms, as this applies only to the rooms of a given building it says nothing about rooms in distinct buildings. Hence the predicate

$$\forall\, r_1 : b_1.rs;\ r_2 : b_2.rs \bullet r_1.ts \cap r_2.ts = \varnothing$$

needs to be conjoined to the predicate of the *Campus* class.

Clearly, capturing the properties of object containment explicitly in this way is cumbersome, particular if the system is large and complex. We would like to be able to give a global invariant that captures directly the condition that distinct rooms anywhere contain distinct terminals, and distinct buildings anywhere contain distinct rooms. The condition that distinct rooms contain distinct terminals, for example, is not an internal invariant of the *Room* class, but rather an invariant of any system containing room objects; nevertheless, it would be convenient to be able to attach such global conditions directly to the *Room* class. A way of doing this is given in Section 7.2.

7.1.2 Example: Russian Dolls

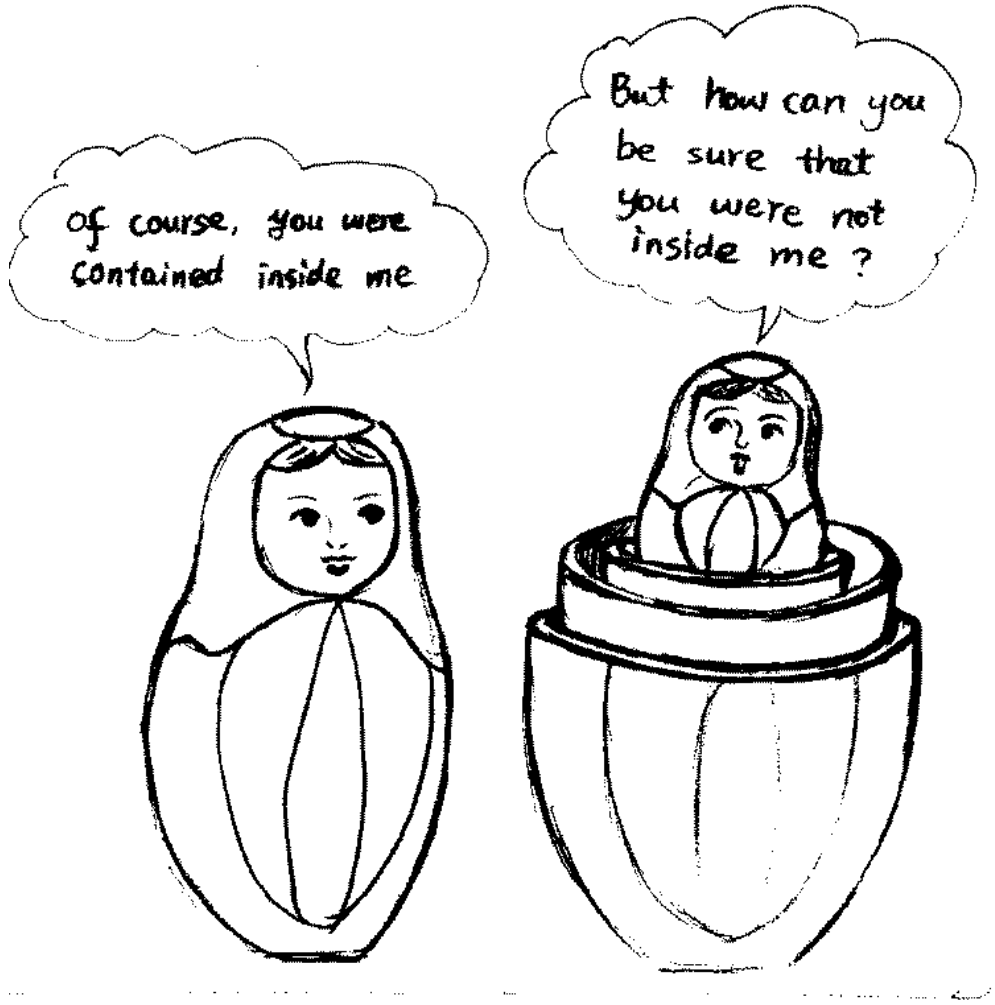


Figure 7.2: Russian Dolls.

Object containment is sometimes an important property of recursive structures. For example, consider the situation of Russian dolls (see Figure 7.2). Each doll is either solid or hollow, and each hollow doll contains another doll that is itself solid or hollow. The fundamental property of this set of recursively embedded dolls is that no doll directly or indirectly contains itself.

An object-oriented specification in Object-Z would be

$$Doll \mathrel{\widehat=} Solid \cup Hollow$$

Solid

$$dolls : \mathbb{P}\, Doll$$

$$dolls = \varnothing$$

Hollow

$$inside : Doll$$
$$dolls : \mathbb{P}\, Doll$$

$$dolls = \{inside\} \cup inside.dolls$$
$$self \notin dolls$$

The attribute *inside* identifies the doll directly contained within a hollow doll; the attribute *dolls* denotes the set of all dolls directly or indirectly contained within a doll. By specifying *Doll* to be the union of the classes *Solid* and *Hollow*, the doll identified by *inside* is either solid or hollow.

The predicate $self \notin dolls$ of the class *Hollow* ensures that a doll does not directly or indirectly contain itself.

This specification takes an object-oriented view, modelling each doll as an object with a unique identity. It could be argued that it is more complex than a functional recursive specification using the free type. However, when object containment is captured by global predicates in Section 7.2, the object-oriented specification mimics in style the definition using free types.

7.2 Capturing the Geometry of Containment

In the examples in the last section, the properties of object containment were formally captured in Object-Z by writing explicit class invariants. There are several consequences of that approach:

- The invariants that result can be complex, particularly as object containment is often a significant aspect of a system's design.
- An appropriate invariant needs to be placed in each relevant class. As the invariant is capturing the same concept of object containment in each case, the specification can become repetitious.

- Only the *properties* that follow from object containment are being captured by the invariant. The geometric notion as to which objects are actually contained within a given object is not explicitly stated.

In this section, specific notation is introduced into Object-Z to capture directly the geometric relationship of object containment. This notation enables the specifier to state explicitly, as part of the class specification, which objects will be directly contained within an object of that class.

To be precise, let each class have an implicitly declared attribute

$$dcontain : \mathbb{P}\,\mathbb{O}$$

where the value of *dcontain* is the set of directly contained objects. Then in any system the relation

$$dcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

introduced in Section 7.1 is determined by

$$\begin{array}{l} \forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ \qquad ob_1 \,\underline{dcon}\, ob_2 \Leftrightarrow ob_2 \in ob_1.dcontain. \end{array}$$

The properties of the relation *dcon* (as stated in Section 7.1) imply invariant conditions on the system that need not be stated explicitly. In terms of the attribute *dcontain* these conditions are:

$$\begin{array}{l} \nexists s : \mathrm{seq}\,\mathbb{O} \bullet \\ \qquad \#s > 1 \\ \qquad \forall\, i : 1 \mathinner{.\,.} \#s - 1 \bullet s(i+1) \in s(i).dcontain \\ \qquad s(1) = s(\#s) \end{array} \qquad [\mathsf{dc1}]$$

(i.e. no object directly or indirectly contains itself)

$$\begin{array}{l} \forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ \qquad ob_1 \neq ob_2 \Rightarrow ob_1.dcontain \cap ob_2.dcontain = \varnothing \end{array} \qquad [\mathsf{dc2}]$$

(i.e. no object is directly contained in two distinct objects). The two predicates dc1 and dc2 are global invariants of any Object-Z specification.

7.2.1 Examples Revisited

With this notation the specification of both the terminal-location and Russian-dolls systems is significantly simplified.

Terminal Location

Terminal

[details of state and operations omitted]

Room

$ts : \mathbb{P}\, Terminal$

$dcontain = ts$

[details of operations omitted]

Building

$rs : \mathbb{P}\, Room$

$dcontain = rs$

[details of operations omitted]

Campus

$bs : \mathbb{P}\, Building$

$dcontain = bs$

[details of operations omitted]

Russian Dolls

$$Doll \mathrel{\widehat{=}} Solid \cup Hollow$$

Solid

$dcontain = \varnothing$

Hollow

$inside : Doll$

$dcontain = \{inside\}$

In both these examples, there are implicit class invariants that follow directly from the properties of *dcontain* stated above. Indeed, from these properties the explicit class invariants given in Sections 7.1.1 and 7.1.2 can be deduced.

The convention is adopted that no mention of *dcontain* need be made if there are no contained objects. Therefore the predicate $dcontain = \varnothing$ can be omitted from the class *Solid*.

7.2.2 A Notational Simplification

If the role of an attribute is to always identify directly contained objects, this can be indicated when the attribute is declared by appending a subscript '$_{\copyright}$' to the appropriate type. This removes the necessity to write an explicit predicate involving *dcontain*. For example, adopting this syntactic convention, the relevant classes in the specification of the terminal location system becomes

Room

$ts : \mathbb{P}\, Terminal_{\copyright}$

Building

$rs : \mathbb{P}\, Room_{\copyright}$

Campus

$bs : \mathbb{P}\, Building_{\copyright}$

while the specification of the Russian-dolls system becomes

$$Doll \mathrel{\widehat{=}} Solid \cup Hollow$$

Solid

Hollow

$inside : Doll_{\copyright}$

The subscript '$_{\copyright}$' is appended to the type of the attribute rather than the attribute itself because the attribute may identify a complex data structure rather than an object reference. For example, the declaration

$$ts : \mathbb{P}\, Terminal_{\copyright}$$

in the *Room* class declares *ts* to be a set, not an object reference. From its type, *ts* is a set of references to objects of class *Terminal*; the $_{\copyright}$ implies these references are to contained objects. Type declarations involving $_{\copyright}$ can be converted into declarations that directly use the attribute *dcontain* instead; for instance:

$$
\begin{array}{|l}
\multicolumn{1}{l}{X} \\
\hline
\begin{array}{|l}
\hline
as : \mathbb{P}\, A_{\copyright} \\
bl : \mathrm{seq}\, B_{\copyright} \\
cf : \mathbb{N} \pfun C_{\copyright} \\
d_1, d_2 : D_{\copyright} \\
e : E \\
fs : \mathbb{P}\, F \\
\hline
\end{array} \\
\hline
\end{array}
$$

is a syntactic shorthand for $\longrightarrow$

$$
\begin{array}{|l}
\multicolumn{1}{l}{X} \\
\hline
\begin{array}{|l}
\hline
as : \mathbb{P}\, A \\
bl : \mathrm{seq}\, B \\
cf : \mathbb{N} \pfun C \\
d_1, d_2 : D \\
e : E \\
fs : \mathbb{P}\, F \\
\hline
dcontain = as \cup (\mathrm{ran}\, bl) \\
\qquad \cup (\mathrm{ran}\, cf) \cup \{d_1, d_2\} \\
\hline
\end{array} \\
\hline
\end{array}
$$

7.2.3 Indirectly Contained Objects

Although the introduction of the attribute *dcontain* is sufficient to capture the above properties of containment, it is often useful to explicitly identify all those objects that a given object directly or indirectly contains. For example, considering the terminal-location system, suppose an operation exists to output the number of terminals which are contained in the campus. The *Campus* class would then be

$$
\begin{array}{|l}
\multicolumn{1}{l}{Campus} \\
\hline
\begin{array}{|l}
\hline
bs : \mathbb{P}\, Building_{\copyright} \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{NumberofTerminals} \\
\hline
num! : \mathbb{N} \\
\hline
num! = \#\{t : Terminal \mid (\exists\, b : bs \bullet \exists\, r : b.rs \bullet t \in r.ts)\} \\
\hline
\end{array} \\
\hline
\end{array}
$$

The predicate of the operation *NumberofTerminals* can be captured more easily if each class has an implicitly declared attribute

$$contain : \mathbb{P}\, \mathbb{O},$$

where the value of *contain* is the set of directly or indirectly contained objects, i.e. in terms of the relation *dcon* introduced in Section 7.1,

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1 \in o_2.contain \Leftrightarrow o_2 \ \underline{dcon^+}\ o_1.$$

To be precise, every class has an implicit class invariant

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

The predicate of the operation *NumberofTerminals* can now be written as

$$num! = \#(contain \cap Terminal).$$

Notice that in terms of the attribute *contain*, the global invariant `dc1` becomes

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

7.2.4 Object Relocation

The geometry of object containment of a system is dynamic because a contained object can relocate from one container to another in the system. For example, considering again the terminal-location system, suppose operations exist to add a terminal to a room, to remove a terminal from a room and to transfer a terminal from one room to another inside a building. The appropriate part of the system specification would then become

Room

$$ts : \mathbb{P}\, Terminal_{©}$$

Add

$$\Delta(ts)$$
$$t? : Terminal$$
$$t? \notin ts \land ts' = ts \cup \{t?\}$$

Remove

$$\Delta(ts)$$
$$t? : Terminal$$
$$t? \in ts \land ts' = ts - \{t?\}$$

Building

$rs : \mathbb{P}\, Room_{©}$

SelectRoom

$r? : Room$

$r? \in rs$

Select2Rooms

$r_1?, r_2? : Room$

$r_1? \neq r_2? \wedge \{r_1?, r_2?\} \subseteq rs$

$Add \mathrel{\widehat=} SelectRoom \bullet r?.Add$

$Remove \mathrel{\widehat=} SelectRoom \bullet r?.Remove$

$Transfer \mathrel{\widehat=} Select2Rooms \bullet r_1?.Remove \parallel r_2?.Add$

It is to be understood that the Δ-list of the operations *Add* and *Remove* in the *Room* class implicitly includes the secondary attributes *dcontain* and *contain* as these values are subject to change whenever the value of the state variable *ts* changes.

Notice that the implicit conditions implied by the geometry of object containment will be maintained at all times[2]. For instance, it will be the case, even though it has not been stated explicitly within the *Add* operation schema that the new terminal added to the room is not already in any other room on the campus.

In some cases, a contained object may be a fixed component of its containing object, i.e. object relocation may not be possible. For example, a room is usually a fixed component of a building. This is implicitly captured by having no operation that can change the attribute *rs* in the class *Building*.

[2]In Z and Object-Z the state invariant must hold before and after each operation.

7.3 Containment in Programming Languages

Some object-oriented programming languages support a view of object containment. In Eiffel, for instance, if the type of an attribute is an *expanded* class, the value of the attribute will be an actual object rather than an object reference. In effect, an attribute of *expanded* class type denotes a contained object. A consequence of this is that although the internal values of a contained object can be updated directly, relocation is not possible: the contained object is treated as a fixed component. Furthermore, both value semantics and reference semantics for objects need to be defined. This contrasts with the approach adopted in this thesis where reference semantics for objects is uniformly maintained. Not only does this simplify the Object-Z semantics, but it permits the relocation of contained objects.

To illustrate this distinction, consider a car that contains four wheel objects and references an owner object. In Eiffel this would be

```
class Car feature
w1, w2, w3, w4 :expanded class Wheel;
owner: Person;
...
end - - class Car
```

while in Object-Z it would be

Car

$$w1, w2, w3, w4 : Wheel_{©}$$
$$owner : Person$$

$$\#\{w1, w2, w3, w4\} = 4$$

...

The *Car* class invariant states that the four wheels are distinct.

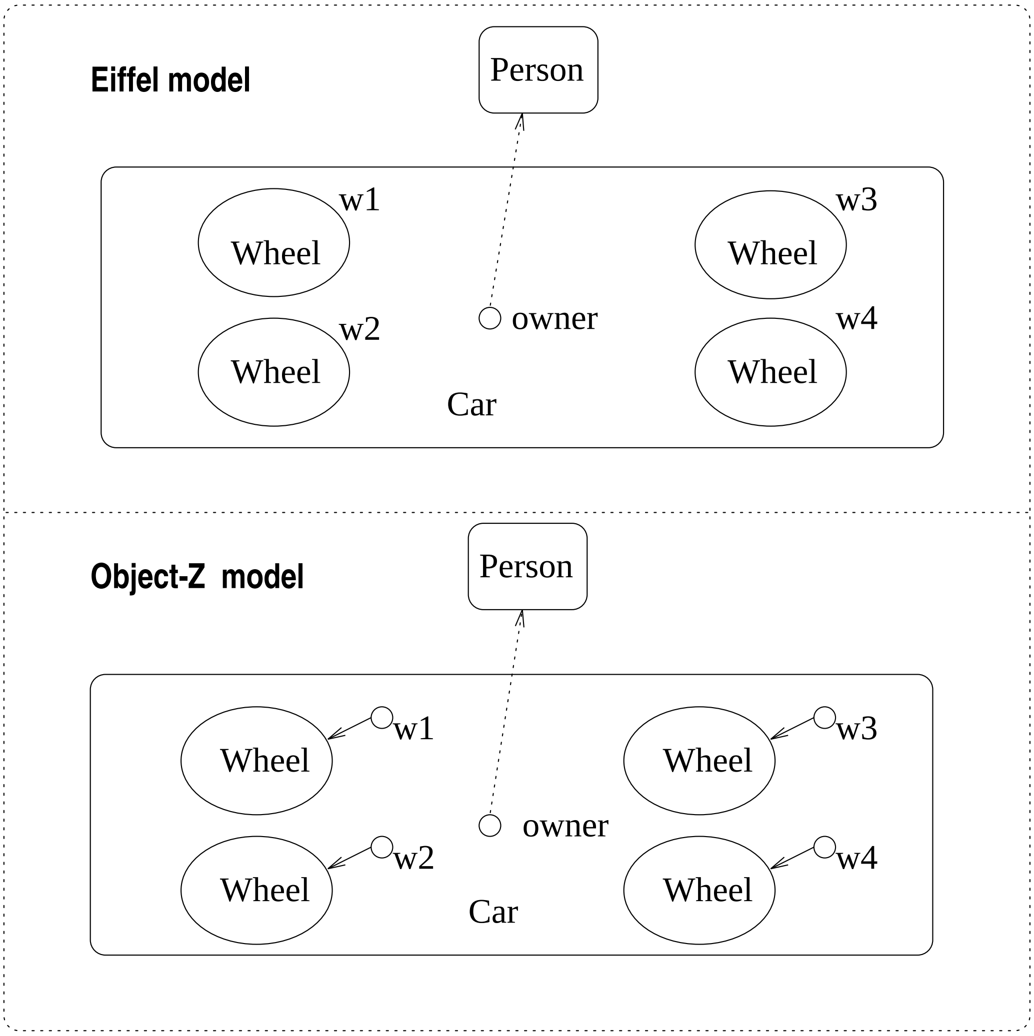


Figure 7.3: Containment in Eiffel and Object-Z

Pictorially, the Eiffel and Object-Z models are given in Figure 7.3, where a dotted or solid arrow denotes an object reference, with the solid arrow denoting a reference corresponding to containment. In Object-Z, operations can be specified to allow a wheel of a car to be switched to a different position or even replaced. However, in Eiffel the use of the *expanded* class type means that such operations are not possible.

The value semantics of the Eiffel *expanded* class type also ensures that no aliasing[3] is possible for any object of such a class. If the source (right hand side) of an assignment

[3]Aliasing occurs when an object can be accessed in more than one way, see [25, 67, 85].

is an object of *expanded* class type then the value of the source object is copied to the target; a reference to the source object is not copied. A consequence is that, in Eiffel, an object is the unique client of any object of expanded class type it contains; in effect, a client has exclusive control of its contained objects. This provides for aliasing protection, although with this approach the notions of control and containment are not distinguished; this issue is discussed further in Section 7.4.

C++ has a notion of object containment similar to that of Eiffel except it makes object containment the default class type and containment does not imply unique control; i.e. C++ does not allow the relocation of a contained object from one container to another, but it does allow a contained object to be referenced by objects other than its containing object. In C++, the car example would be

```
class Car {
Wheel w1,w2,w3,w4;
Person *owner;
...
}
```

7.4 Containment and Access

In object-oriented systems it is sometimes the case that a contained object can be accessed by only the containing object. However, the notions of containment and access are in general quite distinct and should not be confused in system modelling: containment is concerned with the relative geometry of objects; access is concerned with the right of one object to send a message to another object. Often within object-oriented systems the case arises when one object contained within another can be sent messages by a third object elsewhere in the system. This is illustrated in the following example.

7.4.1 Example: A Banking System

A banking system consists of account objects, customer objects and bank objects. An account has a balance, a limit below which the balance must not fall and a set of owners who can operate the account by making deposits or withdrawals. Each account is contained within a bank (i.e. an account is referenced by a unique bank) which is able to change the limit of the account. We shall suppose that an account may be shared between customers. In Object-Z this becomes

Account

$$bal, lim : \mathbb{Z}$$
$$owners : \mathbb{P}\, Customer$$
$$bal \geq lim \wedge owners \neq \varnothing$$

Deposit

$$\Delta(bal)$$
$$n? : \mathbb{N}$$
$$bal' = bal + n?$$

Withdraw

$$\Delta(bal)$$
$$n? : \mathbb{N}$$
$$bal' = bal - n?$$

ChangeLimit

$$\Delta(lim)$$
$$n? : \mathbb{N}$$
$$lim' = n?$$

Customer

$$accs : \mathbb{P}\, Account$$
$$\forall\, a : accs \bullet self \in a.owners$$

$$Deposit \mathrel{\widehat{=}} [a : accs] \bullet a.Deposit$$
$$Withdraw \mathrel{\widehat{=}} [a : accs] \bullet a.Withdraw$$

Bank

$$accs : \mathbb{P}\, Account_{\copyright}$$

$$ChangeLimit \mathrel{\widehat{=}} [a : accs] \bullet a.ChangeLimit$$

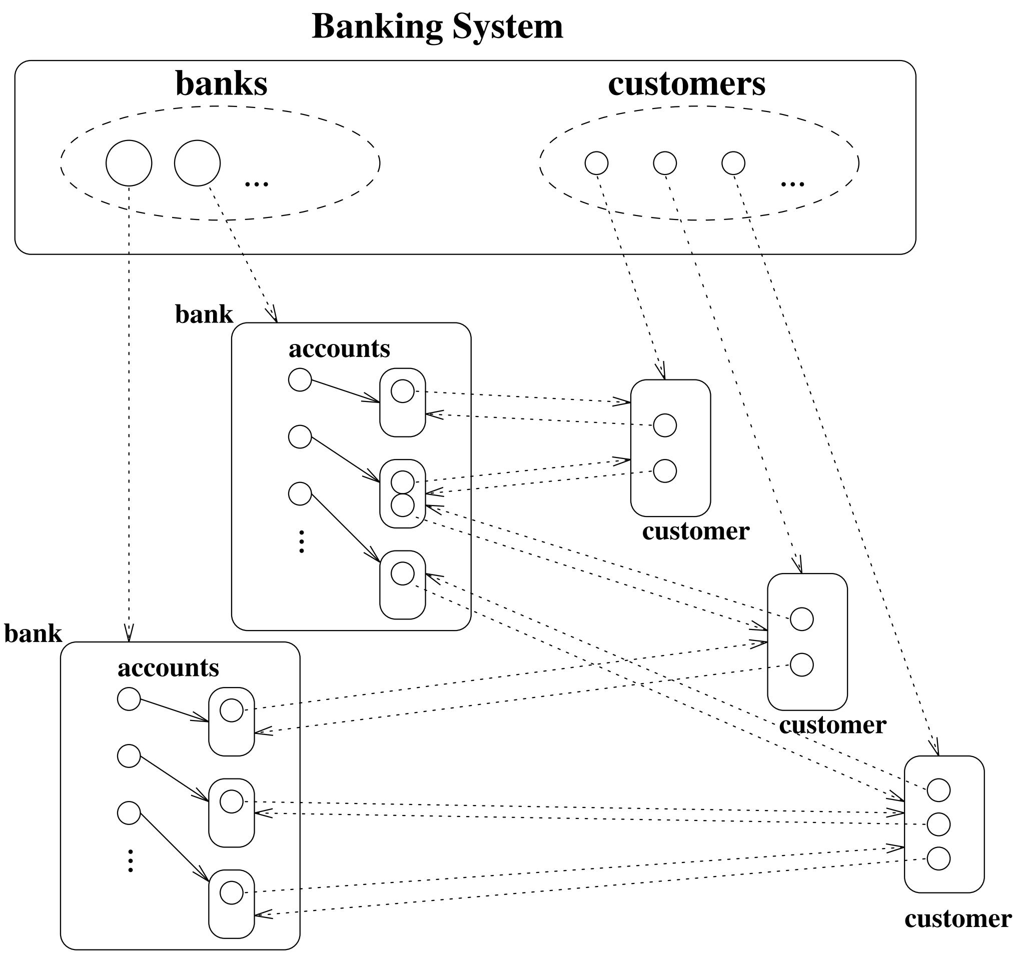


Figure 7.4: A banking system

A banking system consists of a set of banks and a set of customers.

BankingSystem

$$banks : \mathbb{P}\, Bank$$
$$customers : \mathbb{P}\, Customer$$

$$\bigcup\{b : banks \bullet b.accs\} = \bigcup\{c : customers \bullet c.accs\}$$

$$Deposit \mathrel{\widehat=} [c : customers] \bullet c.Deposit$$

$$Withdraw \mathrel{\widehat=} [c : customers] \bullet c.Withdraw$$

$$ChangeLimit \mathrel{\widehat=} [b : banks] \bullet b.ChangeLimit$$

The state invariant of this class ensures that the accounts contained in *banks* are precisely those accounts accessed by *customers*.

Because accounts are contained in banks, the set of accounts is partitioned between the banks, i.e. each account is uniquely associated with a bank. However, customers access the balance of accounts owned by them, while the banks access only the limit of the accounts they contain. The object-reference structure of this banking system is illustrated in Figure 7.4 (where a dotted or solid arrow denotes an object reference, with the solid arrow denoting a reference corresponding to containment).

In Chapter 8, I further discuss the distinction between the notions of object containment and exclusive object control.

7.5 Modelling Shared Containment

The geometric notion of object containment considered so far in this thesis has been that of *unique* containment, i.e. an object cannot be directly contained within two distinct objects. However, this is not the only containment geometry found within real systems. In general, objects may overlap and share contained objects, e.g. two rooms may share a wall in a building. Furthermore, in some systems an object may contain only part of another object, e.g. a street may be located in several suburbs and hence is partially contained by each of the suburbs that share it. That is, property (2) in the early part of Section 7.1 holds for unique containment, but not for a more general notion of shared containment.

The geometric complexities that arise with this notion of shared containment are illustrated in Figure 7.5. In that figure, object s is directly, but only partially, contained and shared by the three objects q, r and t. On the other hand, t is directly and uniquely contained by r. Also, s is indirectly (via t) partially contained by r. The graph on the right hand side of Figure 7.5 captures the geometric relationships between objects q, r, s and t, where a dashed (not dotted) arrow denotes direct shared containment and a solid arrow denotes direct unique containment.

Figure 7.5 suggests three ideas implicit in the notion of shared and unique contain-

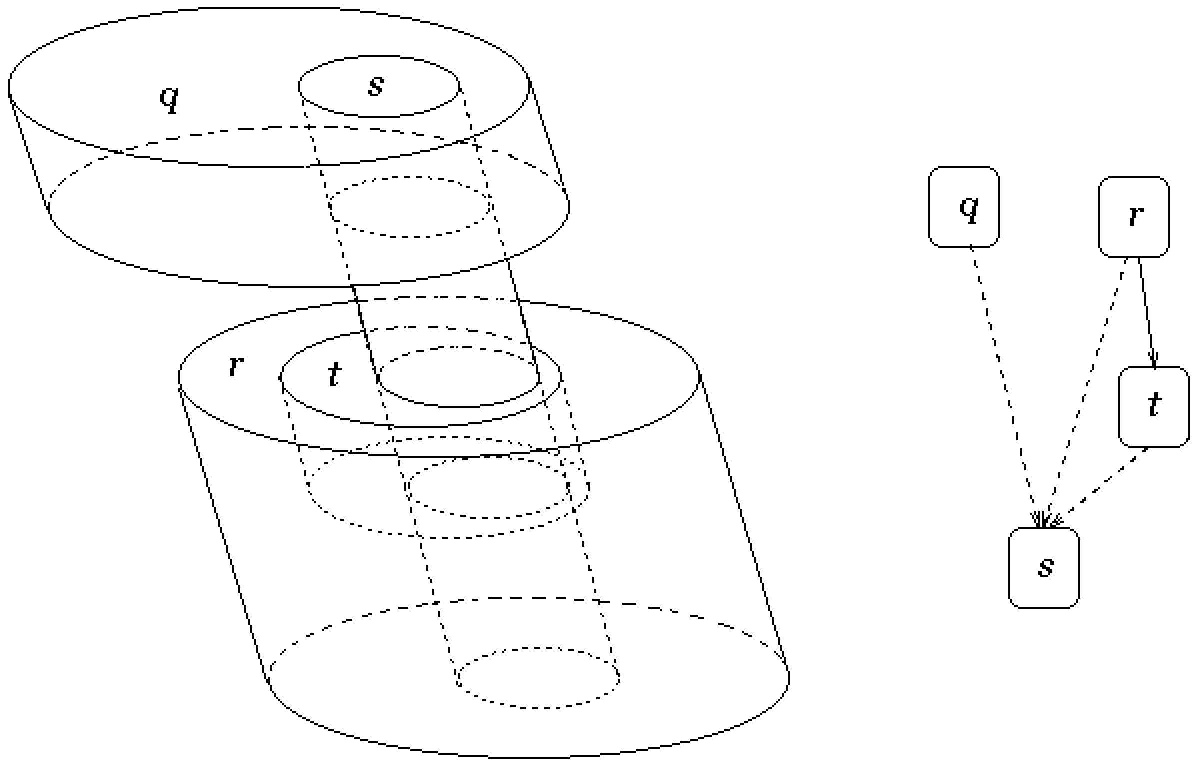


Figure 7.5: The geometry of shared containment

ment:

(1) an object cannot directly or indirectly contain itself, regardless of whether the containment is shared or unique;

(2) an object cannot be directly uniquely contained within two distinct objects (as in Section 7.1); and

(3) for any object, its set of directly contained but sharable objects is disjoint from its set of directly uniquely contained objects.

To capture these ideas formally, let

$$sdcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

denote the relation of direct but sharable containment, i.e.

$$ob_1 \ \underline{sdcon} \ ob_2$$

if and only if object ob_1 directly contains but may share object ob_2. Let DC (direct containment) denote the relation

$$DC : \mathbb{O} \leftrightarrow \mathbb{O}$$

defined to be the union of the two relations $dcon$ (as defined in Section 7.1) and $sdcon$, i.e.

$$DC = dcon \cup sdcon.$$

The three conditions above respectively require that

$$\nexists ob : \mathbb{O} \bullet ob \ \underline{DC^+} \ ob,$$

$$dcon^\sim \in \mathbb{O} \nrightarrow \mathbb{O},$$

$$dcon \cap sdcon = \varnothing.$$

The notion of shared containment can be incorporated into Object-Z in much the same way that unique containment was modelled in Section 7.2.2. Briefly, let every class have two implicit attributes

$$sdcontain, scontain : \mathbb{P}\,\mathbb{O}$$

where $sdcontain$ denotes the set of directly contained but sharable objects, while $scontain$ denotes the directly and indirectly contained but sharable objects. Each class has an implicit invariant

$$\begin{array}{l} scontain = \{ob : \mathbb{O} \mid \\ \qquad ob \notin contain \\ \qquad \exists\, s : \mathrm{seq}_1\, \mathbb{O} \bullet \\ \qquad\qquad s(1) \in (dcontain \cup sdcontain) \\ \qquad\qquad s(\#s) = ob \\ \qquad\qquad \forall\, i : 1 \mathinner{\ldotp\ldotp} \#s - 1 \bullet s(i+1) \in (s(i).dcontain \cup s(i).sdcontain)\} \end{array}$$

In terms of the attributes *dcontain*, *contain*, *sdcontain* and *scontain*, the above conditions on the relations *dcon* and *sdcon* imply the following predicates are implicit invariants of any system:

$$\nexists ob : \mathbb{O} \bullet ob \in (ob.contain \cup ob.scontain),$$

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1 \neq o_2 \Rightarrow o_1.dcontain \cap o_2.dcontain = \varnothing,$$

$$\forall\, ob : \mathbb{O} \bullet ob.dcontain \cap ob.sdcontain = \varnothing.$$

7.5.1 Example: Walls, Rooms and Buildings

Consider once again a campus consisting of buildings and rooms where each room in a building has a set of walls each of which may be shared with some other room. Furthermore, suppose that in the campus the buildings are physically separated so that no wall is shared between different buildings. An Object-Z specification of the campus would include the classes

Wall

…

Room

$ws : \mathbb{P}\, Wall_{\circledS}$

…

Building

$rs : \mathbb{P}\, Room_{\copyright}$

$ws : \mathbb{P}\, Wall_{\copyright}$

$\bigcup\{r : rs \bullet r.ws\} \subseteq ws$

…

where the notation '$\circledS$', like the notation '$\copyright$' introduced in Section 7.2.2, is a syntactic simplification identifying the directly contained but sharable objects. Without this simplification the state schema of the *Room* class would have been

$ws : \mathbb{P}\, Wall$

$sdcontain = ws$

This specification captures explicitly the geometric view that a building uniquely contains both its rooms and its walls, even although these walls may be shared between the rooms. Notice that this specification does not demand that each wall be shared between rooms; rather, it simply indicates that each wall is possibly shared (i.e. is sharable).

7.5.2 Example: Buildings, Streets and Suburbs

It is possible for an object within a system to both uniquely contain some objects and sharably contain others. For instance, consider a town consisting of suburbs, streets and buildings. An Object-Z specification of the town would include the classes

Building

$\ldots$

Street

$\ldots$

Suburb

$bs : \mathbb{P}\, Building_{\copyright}$
$ss : \mathbb{P}\, Street_{\circledS}$

$\ldots$

This specification captures explicitly the geometric view that buildings are uniquely contained within suburbs, whereas streets may be contained but shared between suburbs.

7.5.3 Other Containment Geometries

Although the geometric notion of unique containment, and to a lesser extent that of shared containment, captures, in our experience, the geometry most commonly occurring in real systems, other geometries of object containment are possible. For example, consider the situation illustrated in Figure 7.6 where object m partially contains object n while at the same time n partially contains m.

Although it would be possible to introduce specific notation to formally capture such geometries in Object-Z, as such structures only occur quite rarely in practice it is adequate to capture the properties implied by such geometries explicitly as class invariants (as in Section 7.1) when the need arises. An example is given in Section 7.6, when modelling the abstract structure of a circular doubly-linked list.

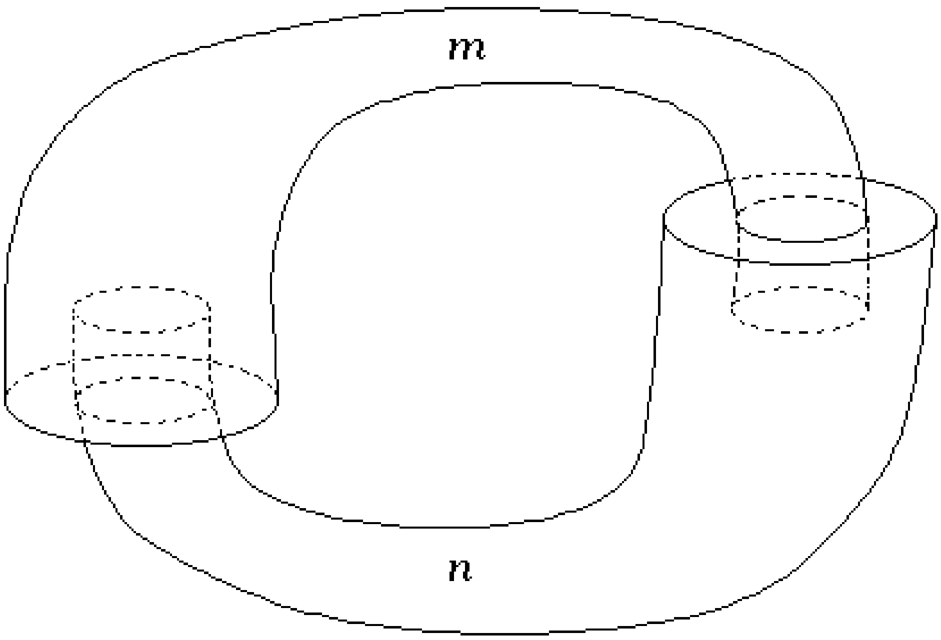


Figure 7.6: The geometry of circular partial containment

7.6 Containment in Abstract Structures

The object references that exist in object-oriented models of abstract recursive structures such as trees or directed acyclic graphs (DAGs) often satisfy the combinatorial properties of object containment, and hence the geometry of object containment can be applied directly when constructing object-oriented models for such structures. As an illustration, consider the following (partial) Object-Z specifications of a tree node and a DAG node where T is a generic type.

$TreeNode[T]$

$value : T$
$subnodes : \mathbb{P}\ TreeNode[T]_{\copyright}$

$DAGnode[T]$

$value : T$
$subnodes : \mathbb{P}\ DAGnode[T]_{\circledS}$

Each node in the tree (or the DAG) has associated with it a value of type T and a (possibly empty) set of subnodes.

The use of object containment in the above examples guarantees a tree (or a DAG) structure without the need to state explicitly as invariants the properties of such a structure.

The notions of unique and shared containment are particularly suitable for modelling acyclic abstract structures, but inadequate for cyclic structures. For instance, an Object-Z specification of a node in a circular doubly-linked list would include

$CDLLnode[T]$

$value : T$
$next, prev : CDLLnode[T]$

$next.prev = self = prev.next$

where the doubly-linked property is captured by an explicit invariant. As an example, consider two instances of the $CDLLnode[T]$ class linked together, i.e. where each one is the *next* (and *prev*) of the other. The (circular) object references between the two instances (nodes) can be viewed as an abstract realisation of the geometry illustrated in Figure 7.6.

7.7 Conclusion

In this chapter, the notion of object containment was captured within a formal framework, first by predicates incorporating the properties of containment within class invariants, and then by extending the Object-Z notation to capture the geometry of object containment directly. The advantage of this extension is that the properties of containment follow implicitly and do not need to be stated explicitly by invariants. Within this formal framework

- reference semantics for objects is sufficient: the introduction of value semantics (e.g. the *expanded* class type of Eiffel) is not needed to capture containment;

- there is a clear distinction between object containment and object access: it is possible to model systems where messages are sent to a contained object by objects other than the containing object; and
- relocation of contained objects is possible without compromising the underlying geometry: the structure of containment is maintained implicitly.

The notion of object containment is particularly useful not only explicitly to capture geometric ideas of object location, but also to specify abstract acyclic structures. In Chapters 9 and 10, the notation of object containment is used to capture the acyclic object reference structure of programming language constructs. By comparison with the specification in Chapter 4, the advantage of applying object containment notation to specify the semantics of programming language is demonstrated.

Chapter 8

Exclusive Object Control within Object Oriented Systems

In this chapter, we are interested in formally modelling a particular association, that of exclusive object control. The exclusive control notion arises when a supplier object has only one direct client object in a system. In this case, the client object has exclusive control over the supplier object. For instance, in a general case, a hand-bag object may be exclusively controlled by a person object. As the notion of exclusive object control is an important aspect of safety and security critical systems, there is the need for specific notations to capture directly such a notion within a formal specification language such as Object-Z. This chapter extends the Object-Z notation to accommodate this need.

The notion of exclusive object control is at first sight apparently similar to the notion of object containment developed in Chapter 7. However, object containment is concerned only with the relative geometric patterns of some common object associations; it is not concerned with the access control aspect of object associations. Therefore, in general it is inappropriate to use the containment notations to capture the notion of exclusive control. Section 8.1 of this chapter details this with an example that motivates the need to extend Object-Z to capture directly the notion of exclusive control. Section 8.2 then presents the specific notation for exclusive object control and details the underlying semantics. Section 8.3 presents a case study to illustrate the role of exclusive control in system modelling. In the case study, exclusive control and containment notations are both used so that the distinction between the two can be demonstrated. Related issues, such as transferring object ownership and the notion of new objects, are also discussed in this case study.

8.1 Capturing Exclusive Control: the Problem

The reference semantics of Object-Z takes the view that every referenced object is potentially shared. There is no implication that distinct object reference declarations introduce distinct object references and hence distinct objects, i.e. declaration $a_1, a_2 : A$ does not imply that a_1 and a_2 reference distinct objects.

However, exclusive control supposes the existence of a unique client (the owner) with exclusive access. The notion of object containment as defined in Chapter 7 can ensure only that no object is directly (uniquely) *contained* in two distinct objects. It indicates nothing about whether objects other than a containing object can *reference* (and hence access) a contained object (The discussion in Chapter 7 clearly demonstrates that containment and access are distinct notions).

8.1.1 Example: Piggy-bank

Suppose each (adult) person has a piggy-bank for storing spare coins (see Figure 8.1).

Furthermore, suppose a married person shares their piggy-bank with their spouse, whereas a single person exclusively owns their piggy-bank. A piggy-bank can be modelled in Object-Z as:

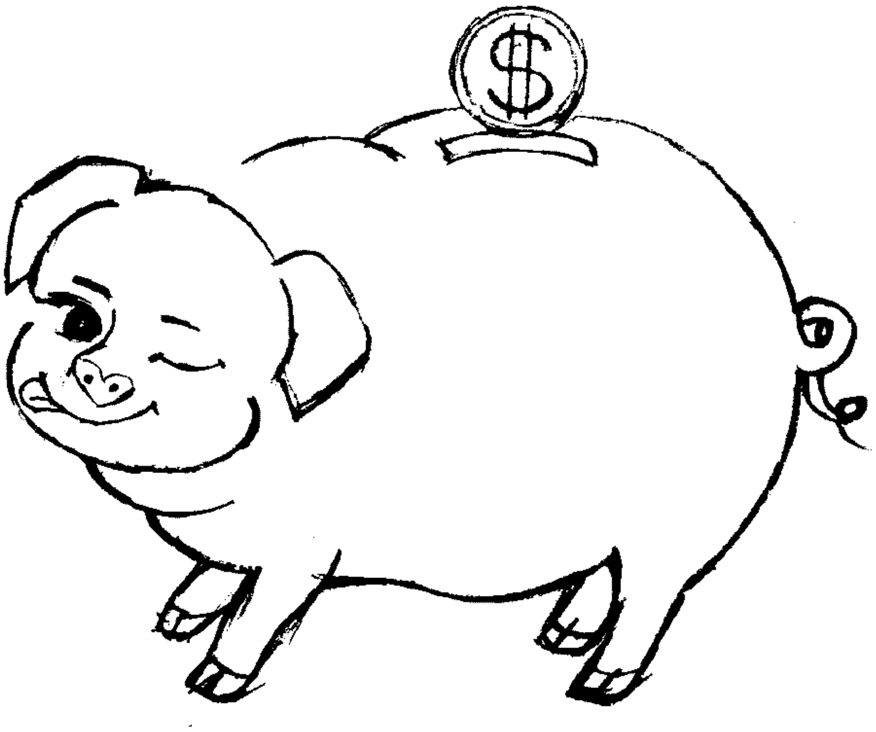


Figure 8.1: Piggy-bank.

PiggyBank

$coins : \mathbb{P}\, Coins$

Init

$coins = \varnothing$

PutIn

$\Delta(coins)$

$in? : Coin$

$in? \notin coins$

$coins' = coins \cup \{in?\}$

EmptyAll

$\Delta(coins)$

$out! : \mathbb{P}\, Coins$

$out! = coins$

$coins' = \varnothing$

where *Coins* denotes the set of all coins.

A specification of married and single people (with their piggy-banks) in Object-Z would be

$$
\begin{array}{|l}
\hline
\textit{MarriedPerson} \\
\begin{array}{|l}
\hline
has_a : PiggyBank \\
spouse : MarriedPerson \\
\hline
spouse \neq self \\
spouse.spouse = self \\
has_a = spouse.has_a \\
\forall\, m : (MarriedPerson \setminus \{self, spouse\}) \bullet m.has_a \neq has_a \\
\hline
\end{array} \\
PutIn \mathrel{\widehat=} has_a.PutIn \\
EmptyAll \mathrel{\widehat=} has_a.EmptyAll \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{SinglePerson} \\
\begin{array}{|l}
\hline
has_a : PiggyBank_{\copyright} \\
\hline
\end{array} \\
PutIn \mathrel{\widehat=} has_a.PutIn \\
EmptyAll \mathrel{\widehat=} has_a.EmptyAll \\
\hline
\end{array}
$$

Notice that the semantics of the containment '$\copyright$' ensures the following property:

$$
\begin{array}{l}
\forall\, s_1, s_2 : SinglePerson \bullet \\
\qquad\qquad s_1 \neq s_2 \Rightarrow s_1.has_a \neq s_2.has_a
\end{array}
$$

which, however, by itself does not prevent an object of *SinglePerson* and an object of *MarriedPerson* referencing the same piggy-bank object. Moreover it is also inappropriate to use the object containment notation to specify the class *MarriedPerson*, such as $has_a : PiggyBank_{\copyright}$ in the state schema of the class *MarriedPerson*, because the semantics of '$\copyright$' will invalidate the property

$$
\begin{array}{l}
\forall\, m : MarriedPerson \bullet \\
\qquad\qquad m.has_a = m.spouse.has_a
\end{array}
$$

(the property which allows a married couple to share the same piggy-bank object.)

To ensure the exclusive control relation between single people and their piggy-banks, it is possible to further impose system constraints on the environment of those *SinglePerson* objects in a system, i.e.

$$
\begin{array}{l}
\textit{System} \\
\hline
\textit{people} : \mathbb{P}(\textit{SinglePerson} \cup \textit{MarriedPerson}) \\
\hline
\forall\, s : \textit{SinglePerson} \bullet \qquad\qquad [\mathtt{i}] \\
\qquad \forall\, m : \textit{MarriedPerson} \bullet s.has_a \neq m.has_a \\
\hline
\textit{PutIn} \mathrel{\widehat=} [p? : \textit{people}] \bullet p?.\textit{PutIn} \\
\textit{EmptyAll} \mathrel{\widehat=} [p? : \textit{people}] \bullet p?.\textit{EmptyAll} \\
\hline
\end{array}
$$

However, capturing the exclusive control property of the class *SinglePerson* explicitly in this way is cumbersome, particular if the specification is complex and contains many objects. Furthermore, if the specification needs to be extended to include a new class *Child*, say, which also has an attribute has_a referencing a piggy-bank object that may be shared with their parents, then the following predicate which is similar to [i] would need to be included in the specification.

$$
\begin{array}{l}
\forall\, s : \textit{SinglePerson} \bullet \\
\qquad\qquad \forall\, c : \textit{Child} \bullet s.has_a \neq c.has_a
\end{array}
$$

This problem indicates that we need a new notation to capture this exclusive control notion directly in the Object-Z specification language.

8.2 Notation for Exclusive Control

In this section, the properties of exclusive control are first formalised, and then notation is introduced into Object-Z to capture directly these properties.

8.2.1 Formalising Exclusive Control

Broadly speaking, if one object references another object, either the object exclusively controls the referenced object and no other object can reference it, or the referenced object is free (or potentially free) to be referenced by other objects.

To capture this idea formally, let

$$ec : \mathbb{O} \leftrightarrow \mathbb{O}$$

denote the relation whereby

$$ob_1 \mathrel{\underline{ec}} ob_2$$

if and only if object ob_1 exclusive controls object ob_2. In contrast, let

$$free : \mathbb{O} \leftrightarrow \mathbb{O}$$

denote the relation whereby

$$ob_1 \mathrel{\underline{free}} ob_2$$

if and only if object ob_1 references ob_2, but ob_2 is free to be referenced by other objects.

The properties of the relations ec and $free$ can be formally captured as

$$\forall\, ob : \mathbb{O} \bullet ec(\!|\{ ob \}|\!) \cap free(\!|\{ ob \}|\!) = \varnothing \qquad \texttt{[ii]}$$

$$\forall\, ob_1, ob_2 : \mathbb{O} \bullet \qquad \texttt{[iii]}$$

$$ob_1 \neq ob_2 \Rightarrow ec(\!|\{ ob_1 \}|\!) \cap (ec \cup free)(\!|\{ ob_2 \}|\!) = \varnothing$$

8.2.2 Object-Z Notation for Exclusive Control

In this subsection, specific notation is introduced into Object-Z to capture directly the notion of exclusive object control. This notation enables the specifier to state explicitly, as part of the class specification, which objects will be exclusively controlled by objects of that class.

Let each class have implicitly declared secondary attributes

$$excon, freerefs : \mathbb{P}\,\mathbb{O}$$

where the value of *excon* is the set of exclusively controlled objects, while that of *freerefs* is the set of referenced objects that are free to be referenced by other objects. Then in any system the relations

$$ec, free : \mathbb{O} \leftrightarrow \mathbb{O}$$

introduced in Section 8.2.1 are determined by

$$\begin{array}{l} \forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ \qquad\qquad ob_1 \,\underline{ec}\, ob_2 \Leftrightarrow ob_2 \in ob_1.excon \\ \forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ \qquad\qquad ob_1 \,\underline{free}\, ob_2 \Leftrightarrow ob_2 \in ob_1.freerefs \end{array}$$

The properties of the relations *ec* and *free* (as stated in Section 8.2.1) imply invariant conditions on the system that need not be stated explicitly. In terms of the attributes *excon* and *free* these conditions are:

$$\forall\, ob : \mathbb{O} \bullet ob.excon \cap ob.freerefs = \varnothing \qquad \texttt{[iia]}$$

$$\begin{array}{l} \forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ \qquad ob_1 \neq ob_2 \Rightarrow ob_1.excon \cap (ob_2.excon \cup ob_2.freerefs) = \varnothing \end{array} \qquad \texttt{[iiia]}$$

The above predicates can be combined and simplified to give the following predicate:

$$\begin{array}{l} \forall\, ob_1, ob_2 : \mathbb{O} \bullet \\ \qquad ob_1.excon \cap ob_2.freerefs = \varnothing \\ \qquad ob_1 \neq ob_2 \;\Rightarrow\; ob_1.excon \cap ob_2.excon = \varnothing \end{array} \qquad \texttt{[iv]}$$

This predicate is a global invariant of any Object-Z specification.

Considering again the piggy-bank example in Section 8.1, using this global invariant the exclusive control relation between single people and their piggy-banks can be ensured by specifying the class *SinglePerson* as:

$$
\begin{array}{|l}
\hline
\textit{SinglePerson} \\
\hline
\quad \begin{array}{|l}
has_a : PiggyBank \\
\hline
excon = \{has_a\} \\
freerefs = \varnothing \\
\hline
\end{array} \\
\ldots \\
\hline
\end{array}
$$

If the role of an attribute is always to identify exclusive controlled objects, this can be indicated when the attribute is declared by appending a subscript '$_{Ⓔ}$' to the appropriate type. This removes the necessity to write explicit predicates involving *excon* and *freerefs*. For example, adopting this syntactic convention, the class *SinglePerson* becomes

$$
\begin{array}{|l}
\hline
\textit{SinglePerson} \\
\hline
\quad \begin{array}{|l}
has_a : PiggyBank_{Ⓔ} \\
\hline
\end{array} \\
\ldots \\
\hline
\end{array}
$$

As object containment notations '$_{©}$' and '$_{Ⓢ}$' do not restrict object access, e.g. '$_{©}$' disallows a contained object to have more than one container but it allows a contained object to have more than one client object, contained objects are classified as part of the free referenced objects, i.e.

$$\forall\, ob : \mathbb{O} \bullet (ob.dcontain \cup ob.sdcontain) \subseteq ob.freerefs$$

where *dcontain* and *sdcontain* denotes the set of objects directly contained in *ob* (see Chapter 7).

Type declarations involving '$_{Ⓔ}$', '$_{©}$' and '$_{Ⓢ}$' can be converted into declarations that directly use the attribute *excon*, *dcontain* and *freerefs* instead; for instance, suppose a class A is defined as:

A

$\ldots$

$OpA \mathrel{\widehat=} \ldots$

Then

B

$as_0 : \mathbb{P}\, A$

$as_1 : \mathbb{P}\, A_{\circledS}$

$as_2 : \mathbb{P}\, A_{\copyright}$

$as_3 : \mathbb{P}\, A_{\textcircled{E}}$

$\ldots$

$\downarrow$

can be converted to $\longrightarrow$

B

$as_0 : \mathbb{P}\, A$

$as_1 : \mathbb{P}\, A$

$as_2 : \mathbb{P}\, A$

$as_3 : \mathbb{P}\, A$

$freerefs = as_0 \cup as_1 \cup as_2$

$sdcontain = as_1$

$dcontain = as_2$

$excon = as_3$

$\ldots$

8.2.3 Preventing Indirect Access

The global class invariant [`iv`] (above) ensures that exclusively controlled objects cannot be directly referenced by objects other than the owner. However, there is an indirect way whereby an object other than the owner can access an exclusively controlled object. For instance, consider the definition of the class C

C

$b : B$

$Op \mathrel{\widehat=} [a : b.as_3] \bullet a.OpA$

where the classes A and B are defined in Section 8.2.2. In effect, an object c of class C can access the exclusively controlled objects of $c.b$. This access path can be blocked if the client object attributes referencing exclusively controlled objects are invisible to the environment. Therefore a general rule would be that

> in every Object-Z class, attributes referencing exclusively controlled objects are hidden from the environment (by not being placed in the class visibility list[1]).

To enforce the above rule in Object-Z specifications, we introduce a class invisibility-list notation '$\not\upharpoonright(...)$' to complement the class visibility-list. For instance, the conversion for the class B above is automatically enhanced with the class invisibility-list. as:

B

$\ldots$

$as_3 : \mathbb{P}\, A_{Ⓔ}$

$\ldots$

$\downarrow$

can be converted to $\longrightarrow$

B

$\not\upharpoonright(as_3)$

$\ldots$

$as_3 : \mathbb{P}\, A$

$\ldots$

$excon = as_3$

$\ldots$

The notation '$\not\upharpoonright(as_3)$' denotes that the attribute a_3 is not in the visibility list of the class B. This invisibility-list mechanism facilitates the notion of exclusive control by ensuring that no objects other than the owner can directly access an owned object through the dot notation.

In the next section, the notation for exclusive object control is applied to specify a computer file system. This case study illustrates the distinction between exclusive control and object containment.

[1]A visibility-list introduces a list of features which are accessible via the dot notation.

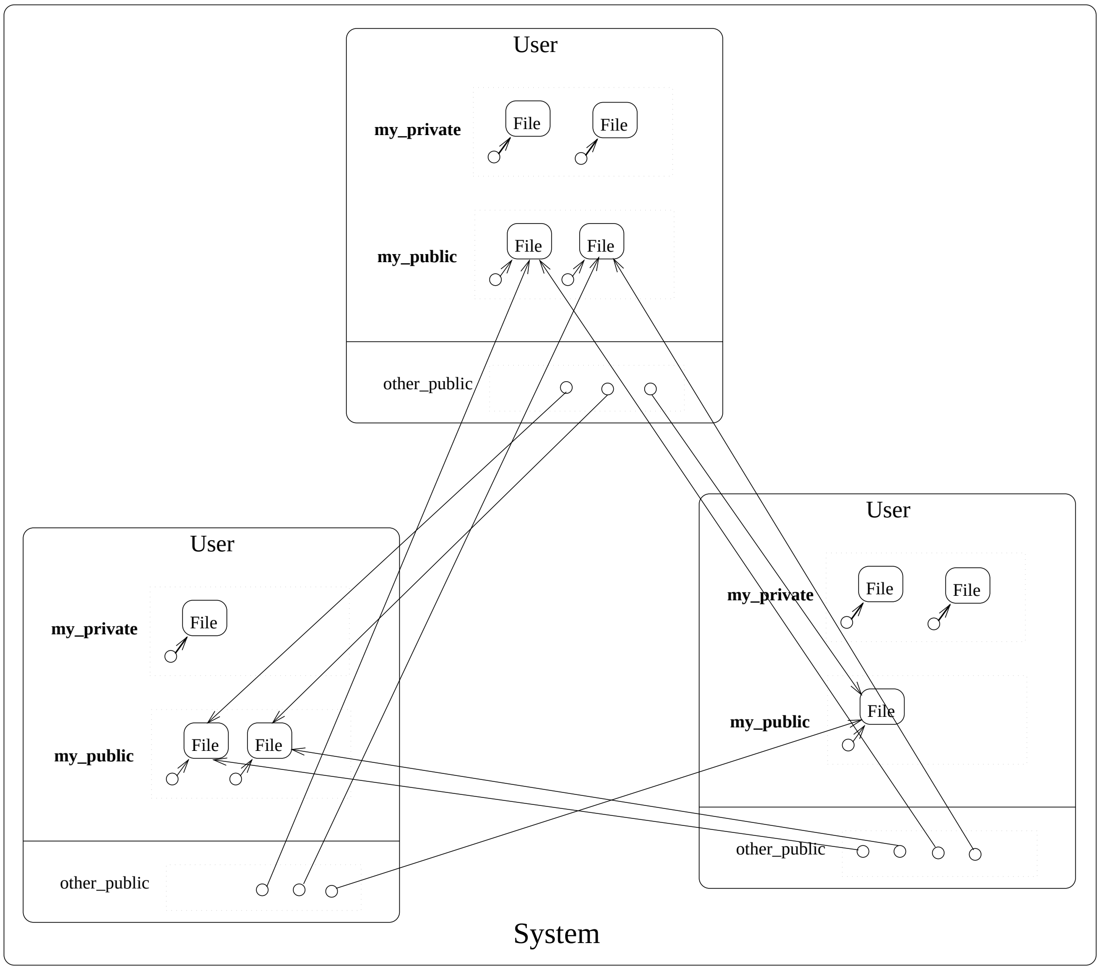


Figure 8.2: Files and Users.

8.3 Case Study: Files and Users

Consider a computer system in which a user can both exclusively own some files, and yet share other files with other users. Any file in the system whether shared or not, has one user as nominal owner. A file can be created or deleted by its nominal owner. Each file of a user can be private or public. If a file is private then no other user can access the file, while if the file is public then all users in the system can access the file. A user can change his or her files from private to public and vice-versa. Private files can also be transferred between users. As an example, an object-reference structure of such a system is illustrated in Figure 8.2. The following is an Object-Z specification of this system.

Firstly, let the skeletal class

File

$$content : Text$$

Update

$$\Delta(content)$$

represent a file, where *Text* is a given type.

A computer user can be modelled as:

User

$$\begin{array}{l} my\_private : \mathbb{P}\, File_{Ⓔ} \\ my\_public : \mathbb{P}\, File_{©} \\ \Delta \\ other\_public : \mathbb{P}\, File \end{array}$$

AddToMyPriv

$$\begin{array}{l} \Delta(my\_private) \\ f? : File \\ \hline f? \notin my\_private \\ my\_private' = my\_private \cup \{f?\} \end{array}$$

DelFromMyPriv

$$\begin{array}{l} \Delta(my\_private) \\ f? : File \\ \hline f? \in my\_private \\ my\_private' = my\_private \setminus \{f?\} \end{array}$$

AddToMyPub

$$\begin{array}{l} \Delta(my\_public) \\ f? : File \\ \hline f? \notin my\_public \\ my\_public' = my\_public \cup \{f?\} \end{array}$$

DelFromMyPub

$$\begin{array}{l} \Delta(my\_public) \\ f? : File \\ \hline f? \in my\_public \\ my\_public' = my\_public \setminus \{f?\} \end{array}$$

$$OpenAccess \mathrel{\widehat=} DelFromMyPriv \wedge AddToMyPub$$

$$LimitAccess \mathrel{\widehat=} DelFromMyPub \wedge AddToMyPriv$$

$$Select \mathrel{\widehat=} [f? : File \mid f? \in (my\_private \cup my\_public \cup other\_public)]$$

$$Update \mathrel{\widehat=} Select \bullet f?.Update$$

The secondary attribute *other_public* denotes all other public files which are not owned by a user but can be accessed by the user. For any user object x, the value of $x.other\_public$ is dependent on the environment of x. The precise meaning of this attribute is defined by the state invariant of the system class below.

The system consists of a set of users.

$$
\begin{array}{l}
\textit{System} \\
\quad users : \mathbb{P}\, User \\
\quad \hline \\
\quad \forall u_1 : users \bullet \\
\qquad u_1.other\_public = \bigcup \{ u_2 : users \mid u_2 \neq u_1 \bullet u_2.my\_public \} \\[1em]
\quad \textit{Select} \\
\quad u? : User \\
\quad \hline \\
\quad u? \in users \\[1em]
\quad \textit{SelectTwo} \\
\quad u_1?, u_2? : User \\
\quad \hline \\
\quad u_1? \neq u_2? \wedge \{u_1?, u_2?\} \subseteq users \\[1em]
\quad Update \mathrel{\widehat{=}} Select \bullet u?.Update \\
\quad Create \mathrel{\widehat{=}} Select \bullet u?.AddToMyPriv \\
\quad Delete \mathrel{\widehat{=}} Select \bullet u?.DelFromMyPriv \\
\quad OpenAccess \mathrel{\widehat{=}} Select \bullet u?.OpenAccess \\
\quad LimitAccess \mathrel{\widehat{=}} Select \bullet u?.LimitAccess \\
\quad Transfer \mathrel{\widehat{=}} SelectTwo \bullet u_1?.DelFromMyPriv \parallel u_2?.AddToMyPriv
\end{array}
$$

The state invariant of *System* implies that if a file is public then all users in the system can access the file.

The operation $Create \mathrel{\widehat{=}} Select \bullet u?.AddToMyPriv$ specifies that a file $f?$ can be added to a user $u?$. Although the specification is not concerned with how the file

$f?$ is created, it precisely captures that the file added to the user is a new file which is not referenced by any user in the system before the operation is invoked. This is because of the explicit precondition $f? \notin u?.my\_private$ and the global class invariant ([iv] in Section 8.2.2) ensuring that no users other than $u?$ can reference the files of $u?.my\_private$.

The operations *OpenAccess* and *LimitAccess* specify that a file $f?$ of a user $u?$ can be transferred between $u?.my\_private$ and $u?.my\_public$. These two operations illustrate that a client object can change its exclusive control of another object.

The operation *Transfer* specifies that a private file can be transferred from one user to another. This operation illustrates that the unique access right of an object can be transferred from one owner to another.

8.4 Conclusion and Discussion

Object sharing (aliasing) and object control are both important notions in object-oriented systems, and the formal specification of such systems needs to capture precisely these notions. The reference semantics of Object-Z takes the view that every referenced object is potentially shared. In this thesis, we have extended the Object-Z notation to directly capture the notion of exclusive object control. A possible way to implement the exclusive control notion in object-oriented programming languages, such as Eiffel, is to use the *expanded* class type.

General Discussion

Before we begin the next two chapters dealing with programming language semantics, it is worth clarifying the relationship between the general object-oriented concepts of encapsulation, information hiding and aggregation and the corresponding Object-Z constructs. In Object-Z, encapsulation is supported by the class construct which ensures that the state of an object can only be changed by applying a method of the class to the object. Information hiding is supported by the class visibility list. Finally, object aggregation (i.e. client-supplier associations) is supported in Object-Z by class instantiation (i.e. $a : A$ or $as : \mathbb{P} A$ where A is a class). Object containment and exclusive control add various degrees of encapsulation to the object client-supplier associations. For instance, the property of object containment '$\copyright$' prohibits a contained object from having more than one container but it allows a contained object to have more than one client object. The property of exclusive control prohibits an exclusively controlled object from having more than one client object. Exclusive control eliminates any potential for object sharing. This fundamental difference determines that the contained objects and exclusively controlled objects are disjoint. However, the property of exclusive control implies the geometric property of object containment. Although in general object-oriented systems, partial control is more common than exclusive control (e.g. an bank can control the limit of an account and a customer can control the balance of an account, as demonstrated in Chapter 7), the notion of exclusive object control is an important aspect of safety and security critical systems. With the specific notations defined in this thesis, Object-Z is a suitable language to specify safety and/or security critical systems.

Chapter 9

A Block Structured Procedural Language Definition

In Chapter 4, an object-oriented approach has been applied to the specification of a simple programming language semantics. Using an object-oriented style to specify a programming language not only leads to a concise specification, but also improves the extendibility of the semantic representation of the programming language. In the last section of Chapter 4, we identified and discussed some weaknesses of that early version of the object-oriented specification of the programming language and these discussions motivated the development of a generalised polymorphic construct, class-union (Chapter 5), and the development of notations for capturing object containment in Object-Z (Chapter 7). In this chapter, we apply these new extensions, class-union and object containment, to specify a block structured programming language semantics. The block structured programming language is an extension to the simple programming language in Chapter 4 so that the semantic representations in both chapters can be compared. Furthermore, the block structured programming language includes procedures with both value and reference parameter substitution which adds a level of complexity to the language semantics.

9.1 Concrete Syntax of the Language

The additional features of the block structured language over the simple language in Chapter 4 are: a while loop statement, a procedure, a procedure-call statement, and a scope rule similar to Pascal.

The concrete syntax of the block structured programming language is:

$$
\begin{array}{rcl}
\textit{Program} & \rightarrow & \text{`Program'}\ \textit{Id Block} \\
\textit{Id} & \rightarrow & \text{...(as in Section 4.2.1)} \\
\textit{Block} & \rightarrow & \text{`dec:'}\ \textit{Declist Procedures}\ \text{`begin'}\ \textit{Stmt}\ \text{`end'} \\
\textit{Declist} & \rightarrow & \text{...(as in Section 4.2.1)} \\
\textit{Procedures} & \rightarrow & \epsilon \mid \textit{Procedure}\ \text{Procedures} \\
\textit{Procedure} & \rightarrow & \text{`Procedure'}\ \textit{Id}\ \text{`('}\ \textit{Declist}\ \text{`var:'}\ \textit{Declist}\ \text{`);'}\ \textit{Block} \\
\textit{Stmt} & \rightarrow & \text{...(as in Section 4.2.1)} \mid \textit{WhileStmt} \mid \textit{CallStmt} \\
\textit{WhileStmt} & \rightarrow & \text{`while'}\ \textit{Exp}\ \text{`do'}\ \textit{Stmt} \\
\textit{CallStmt} & \rightarrow & \textit{Id}\ \text{`('}\ \textit{Explist Varlist}\ \text{`);'} \\
\textit{Exp} & \rightarrow & \text{...(as in Section 4.2.1)} \\
\textit{Explist} & \rightarrow & \epsilon \mid \textit{Exp}\ \text{`,'}\ \textit{Explist} \\
\textit{Varlist} & \rightarrow & \epsilon \mid \textit{Var}\ \text{`,'}\ \textit{Varlist}
\end{array}
$$

9.2 Semantics of the Language

The semantics of the language is represented in the following order: the model of expressions, the model of statements and the model of procedures and programs.

9.2.1 Expressions

An expression is either a constant, a variable reference, a plus-expression, a less-expression or an and-expression according to the concrete syntax of expressions in Section 4.2.1. Expressions are specified as a class-union:

$$\textit{Exp} \mathrel{\hat{=}} \textit{Const} \cup \textit{Varef} \cup \textit{PlusExp} \cup \textit{LessExp} \cup \textit{AndExp}$$

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

As the acyclic object reference structure of an expression can be specified by the containment notations (introduced in Chapter 7), then the attribute '*exps*' in the class *BaseExp* (of Chapter 4) can be removed, i.e.

BaseExp

$type : Scaltype$
Δ
$val : Scalval$

$type\_of(val) = type$

OutVal

$val! : Scalval$

$val! = val$

where the function *type_of* is defined in Section 4.1.3.

The classes *Const* and *BinaryExp* (in Chapter 4) can be simplified by using the containment notation and the class-union *Exp* as:

Const

BaseExp

$content : Scalval$

$val = content$

BinaryExp

BaseExp

$e_1, e_2 : Exp_{Ⓢ}$

The subscript 'Ⓢ' (sharable containment) appended to the type of e_1 and e_2 specifies that the object reference structure of an expression is acyclic (e.g. an expression cannot be a sub-expression directly or indirectly of itself).

We extend the definition of the class *Varef* (in Chapter 4) to have the additional operations *OutLoc* and *ReLoc* to facilitate the specification of the reference parameter substitutions in a procedure call statement (defined later).

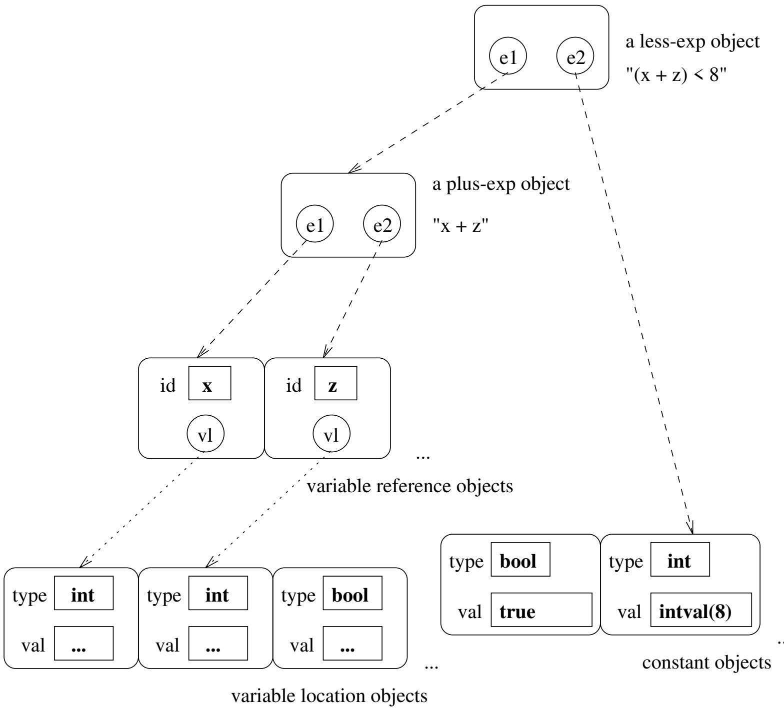


Figure 9.1: Reference Structure of a *Less* Expression.

Varef

$BaseExp$

$id : Id$
$vl : VarLoc$
Δ
$type : Scaltype$
$val : Scalval$

$type = vl.type$
$val = vl.val$

OutLoc

$vl! : VarLoc$

$vl! = vl$

ReLoc

$\Delta(vl)$
$vl? : VarLoc$

$vl' = vl?$

$Assign \mathrel{\widehat=} vl.Assign$

Classes $VarLoc$, $PlusExp$, $LessExp$ and $AndExp$ in Chapter 4 remain unchanged here. Figure 9.1, as an example, illustrates the object structure of a less-expression.

9.2.2 Statements

As the role of the abstract class *BaseStmt* in Chapter 4 is to capture the polymorphic type structure and acyclic object reference structure of a statement, by using class-union and object containment notation, we can eliminate this abstract class here.

A statement is an *assignment* statement, an *if* statement, a *sequence* of statements, a *while* statement or a *call* statement. Statements are specified as a class-union:

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

where the classes *AssignStmt*, *IfStmt* and *SeqStmt* (in Chapter 4) can be simplified by using class-union and containment notations as:

AssignStmt

$$\begin{array}{l} lhs : Varef_{Ⓢ} \\ rhs : Exp_{Ⓢ} \\ \hline lhs.type = rhs.type \end{array}$$

$$Execute \mathrel{\widehat=} rhs.OutVal \parallel lhs.Assign$$

The object reference structure of an *assignment* statement is illustrated by Figure 9.2.

IfStmt

$$\begin{array}{l} test : Exp_{Ⓢ} \\ then, else : Stmt_{Ⓢ} \\ \hline test.type = bool \end{array}$$

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

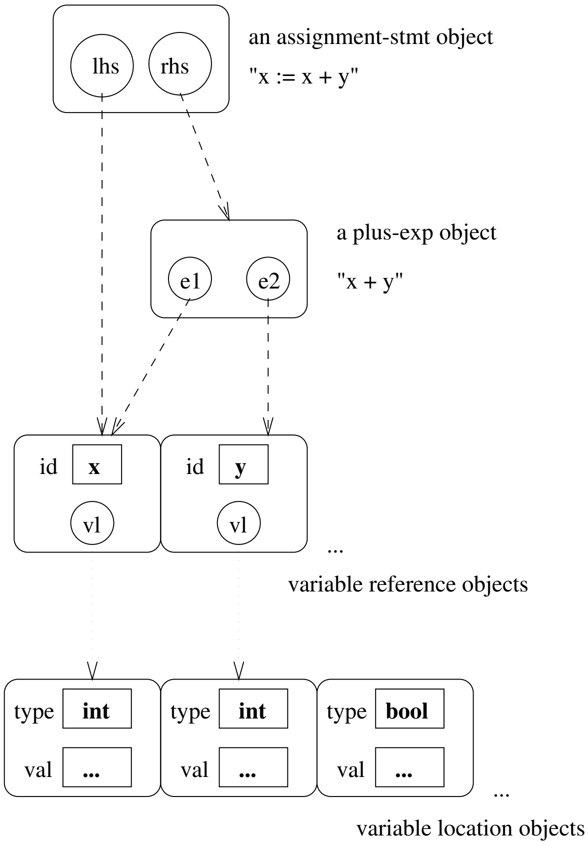


Figure 9.2: Reference Structure of a *Assignment* Statement.

SeqStmt

$stmts : \mathrm{seq}\, Stmt_{\circledS}$

$Execute \mathrel{\widehat=} \fatsemi\ i : 1 \mathinner{\ldotp\ldotp} \#stmts \bullet stmts(i).Execute$

A *while* statement consists of a boolean expression and a body statement. It is modelled as an object of the class *WhileStmt*:

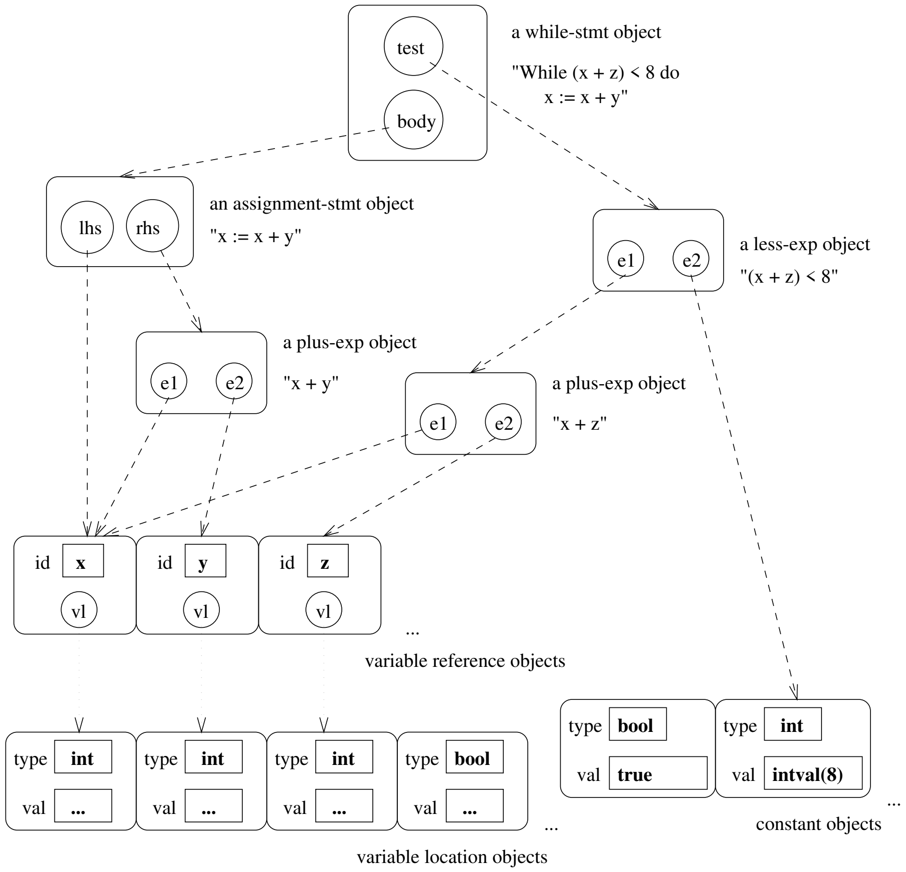


Figure 9.3: Reference Structure of a *While* Statement.

WhileStmt

$test : Exp_{Ⓢ}$

$body : Stmt_{Ⓢ}$

$test.type = bool$

$$Execute \mathrel{\widehat=} ([boolval^{\sim}(test.val)] \land body.Execute) \mathbin{⨾} Execute$$
$$\quad\quad\quad\quad\quad \sqcap$$
$$\quad\quad\quad\quad\quad [\neg\, boolval^{\sim}(test.val)]$$

The object reference structure of a *While* statement is illustrated by Figure 9.3 (which is a merger of Figure 9.1 and Figure 9.2).

The Ease of Reusability

Reusability of the semantic model is readily achieved by using inheritance. For instance, a *repeat* statement class can be defined by inheriting the *while* statement class.

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

A *call* statements consists of a procedure reference, an actual value parameter list and an actual reference parameter list. It is modelled as an object of the class *CallStmt*:

$$
\begin{array}{|l}
\hline
\textit{CallStmt} \\
\quad pr : \textit{Procref}_{ⓢ} \\
\quad avpl : \mathrm{seq}\ \textit{Exp}_{ⓢ} \\
\quad arpl : \mathrm{seq}\ \textit{Varef}_{ⓢ} \\
\quad \hline
\quad \#avpl = \#pr.p.fvpl \wedge \#arpl = \#pr.p.frpl \\
\quad \forall\, i : \mathrm{dom}\ avpl \bullet avpl(i).type = pr.p.fvpl(i).type \\
\quad \forall\, j : \mathrm{dom}\ arpl \bullet arpl(j).type = pr.p.avpl(j).type \\
\textit{ValSubs} \mathrel{\widehat{=}} \bigwedge i : \mathrm{dom}\ avpl \bullet avpl(i).\textit{OutVal} \parallel pr.p.fvpl(i).\textit{Assign} \\
\textit{RefSubs} \mathrel{\widehat{=}} \bigwedge i : \mathrm{dom}\ avpl \bullet arpl(i).\textit{OutLoc} \parallel pr.p.frpl(i).\textit{ReLoc} \\
\textit{Execute} \mathrel{\widehat{=}} (\textit{ValSubs} \wedge \textit{RefSubs}) \mathbin{;} pr.\textit{Execute} \\
\hline
\end{array}
$$

Note that the definition of *Procref* is specified in the following sub-section. The state invariants of the *CallStmt* class specify that

- the length of the actual parameter list from a call statement is equal to the length of the formal parameter list of the called procedure.

- The type of an actual parameter matches to the type of the corresponding formal parameter.

The operation *ValSubs* represents the value substitution from the actual value (expression) parameters of a procedure call to the formal value (variable) parameters of the procedure. The operation *RefSubs* represents the relocation of the variable locations (vl) of the reference variable parameters. Notice that the messages which pass between the parallel operator $\|$ in the operation *ValSubs* are the values of the actual parameters while in the operation *RefSubs* the messages are the object references (identities) of the actual parameters. Such a distinction is made clear in the Object-Z representation. The meaning of a *call* statement is specified by the operation *Execute* which is the meaning of the calling procedure ($pr.Execute$) after the parameter substitutions (*ValSubs* and *RefSubs*).

9.2.3 Procedure and Program

The notion of the scope of variables and procedures is an essential feature of block-structured programming languages. We choose scope rules for the example language similar to those for Pascal: that is, when a variable or a procedure is declared to be inside a subprogram, the declaration has no effect outside the subprogram; the variable or the procedure is *local* to the sub-program. On the other hand, a variable or a procedure declared in the top program may be used anywhere; they are *global*. In brief, the scope rules of the example program are summarised as:

- A statement of the top program can access only the global variables and procedures.
- A statement of a procedure P can access all the variables and procedures which are declared locally in the procedure P. In addition the statement can also access all the variables and procedures which can be accessed by the parent procedure (or program) of P.

A procedure and a program are similar and they both are modelled as objects. Define the class *Block* as the common part of a procedure and a program to be inherited by the procedure class and the program class. A block contains local variable declarations, *vdecs*, and sub-procedures, *subprocs*. It also contains a list of local variable references, *lvars*, and a body statement, *stmt*.

Block

$$
\begin{array}{ll}
vdecs : Id \rightarrowtail\!\!\!\!\!\!\rightarrow VarLoc_{©} & \\
subprocs : Id \rightarrowtail\!\!\!\!\!\!\rightarrow Proc_{©} & \\
lvars : \mathrm{seq}\ Varef_{Ⓢ} & \\
stmt : Stmt_{Ⓢ} & \\
\Delta & \\
vscope : Id \nrightarrow VarLoc & \\
pscope : Id \rightarrowtail\!\!\!\!\!\!\rightarrow Proc & \\
\hline
\forall i : \mathrm{dom}\, subprocs \bullet subprocs(i).parent = self & [\mathsf{pred1}] \\
\#\{v : (Varef \cap scontain) \bullet v.id\} = \#(Varef \cap scontain) & [\mathsf{pred2}] \\
\{vr : (Varef \cap scontain) \bullet vr.id \mapsto vr.vl\} \subseteq vscope & [\mathsf{pred3}] \\
\{p : (Procref \cap scontain) \bullet p.id \mapsto p.proc\} \subseteq pscope & [\mathsf{pred4}]
\end{array}
$$

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

The attributes *vdecs* and *subprocs* are both one-one mappings from identifiers to variable locations or procedures. This provides the unique association of each identifier with its corresponding variable location or procedure in each block. The object containment notation '©' is used to specify the local scope of each block. The whole scope of a block is specified by the value of the secondary variables, *vscope* and *pscope*. The precise meaning of those variables depends on whether the block is inherited as a procedure or a program; therefore only the signature of the *vscope* and *pscope* appear in the class *Block*. (Note that the implicit declared secondary attribute *scontain* was defined in Section 7.5.)

The class invariant pred1 specifies the *parent child* relationships between parent procedures and their sub-procedures. The pred2 specifies that there are no two distinct variables having the same name in a block. The class invariants pred3 and pred4 specify that applied occurrences of the variable and the procedure references are bounded in the scope. As both variables *vscope* and *pscope* are functional mappings **pred3** and **pred4** also specify that any two common named variable references or any two common named procedure references appearing in any statement point at the same location.

The procedure class *Proc* is defined by inheriting the class *Block*, and therefore contains all the information which is specified in the class *Block*. In addition, a *Proc* object consists of a formal value parameter list, *fvpl*, a formal reference parameter list, *frpl*, and a parent procedure or program, *parent*.

Proc

Block

$fvpl : \text{seq}\ Varef_{Ⓢ}$
$frpl : \text{seq}\ Varef_{Ⓢ}$
$parent : Proc \cup Prog$

$vscope = parent.vscope \oplus (vdecs \cup \{vr : \text{ran}\, frpl \bullet vr.id \mapsto vr.vl\})$
$pscope = parent.pscope \oplus subprocs$
$\#\{vr : \text{ran}(fvpl \cup frpl \cup lvars) \bullet vr.id\} = \#fvpl + \#frpl + \#lvars$ [pred5]
$\{vr : \text{ran}(fvpl \cup lvars) \bullet vr.id \mapsto vr.vl\} = vdecs$ [pred6]

The whole access scope of a procedure is defined as the scope of the parent procedure or program overridden by the local variable and procedure declarations. The variables *vscope* and *pscope* in a procedure essentially play the role of passing the static environment and the dynamic environment from parent procedure or program to its sub-procedure. (Compare with scope in VDM or Z specifications [71, 58].)

In the class *Proc*, the predicate **pred5** specifies that any two different parameters or local variables have different identifiers. The predicate **pred6** specifies that the variable locations which are identified by the value parameters and local variables of a procedure are encapsulated (geometrically contained) by the procedure object.

A procedure reference is defined similar to the variable reference as:

$$
\begin{array}{|l}
\hline
\textit{Procref} \\
\begin{array}{|l}
\hline
id : Id \\
p : Proc \\
\hline
\end{array} \\
Execute \mathrel{\widehat=} p.Execute \\
\hline
\end{array}
$$

The program class *Prog* is also defined by inheriting *Block*. In addition, it has a name, *id*.

$$
\begin{array}{|l}
\hline
\textit{Prog} \\
Block \\
\begin{array}{|l}
\hline
id : Id \\
\hline
vscope = vdecs \\
pscope = subprocs \\
\{ vr : lvars \bullet vr.id \mapsto vr.vl \} = vdecs \\
\hline
\end{array} \\
\hline
\end{array}
$$

This completes the language semantics.

From the definition of the class *Prog* (after expansion),

- the abstract syntax of a program is represented as the attributes *vdecs* (declarations), *subprocs* (sub-procedures) and *stmt* (body);
- the static semantics is specified by the class invariant;
- the dynamic semantics is specified by the operation *Execute*.

9.3 Conclusion

In this chapter we applied class-union and object containment constructs to the semantics of a block structured programming language.

The class-union construct was used to precisely define the polymorphic type declaration, such as '*Exp*' and '*Stmt*', in the language model. In contrast to the language model in Chapter 4, where abstract classes, such as *BaseExp* and *BaseStmt*, must be introduced to form a polymorphic inheritance hierarchy to facilitate the required polymorphic type declarations. Then these abstract classes must be excluded by a class invariant in each individual class using such a declaration.

Shared object containment 'Ⓢ' precisely and concisely specified the acyclic object reference structure of expressions and statements in the language model of this chapter, in contrast to the language model in Chapter 4, where the acyclic object reference structure of expressions and statements must be specified by the repetitious class invariants. Furthermore, object containment '©' was used in this chapter to specify the local scope of a procedure or a program precisely.

In summary, the object-oriented semantic model was significantly simplified and improved by using class-union and object containment notions.

The next chapter will apply this object-oriented semantic approach to the object-oriented programming language paradigm itself.

Chapter 10

Modelling Object Oriented Programming Languages

In this chapter, we apply the object-oriented approach to the specification of the semantics of object-oriented programming languages (OOPLs).

The denotational semantics of object-oriented programming languages has been studied and developed. Wolczko[114] and Kamin[72] presented denotational semantics of the Smalltalk programming language. The denotational semantics of inheritance has been investigated by Cook and Palsberg[23] based on Cardelli's work[17]. Others [64, 95] have also contributed to the research work on the semantics of OOPLs. The denotational semantics of the parallel object-oriented programming language POOL has also been studied in [2].

Our approach to the semantics of OOPLs is quite different. The key idea behind this approach is to use a class to model each language construct of OOPLs (in a similar way to Chapter 9). That is, each language construct of OOPLs, such as expressions, statements, methods and classes, and even run-time program entities such as objects (instances of program classes), are modelled by Object-Z classes, whereas traditionally they have been modelled as more primitive mathematical entities such as mappings or composite types (in VDM terms). The aim of our approach is to achieve a more structured and extendible semantics.

In Section 10.1, the object-oriented approach used to model OOPLs is outlined. In Section 10.2, a small OOPL, called SOPL (simple object programming language), is specified in Object-Z and then in Section 10.3 the specification is extended to accommodate more language constructs and demonstrate the extendibility of the model.

To distinguish between the terminology (such as 'class' and 'object') used in both an OOPL (the language being modelled) and in Object-Z (the meta language of the model), the sans serif type style is used to express terms in OOPLs, e.g. class refers to the term 'class' in the context of OOPLs, rather than in the context of Object-Z.

10.1 An OO Approach to Modelling OOPLs

If we take an object-oriented approach to specify OOPLs, then an object-oriented program can be modelled as an object which consists of a collection of static objects, such as expressions, methods, statements and classes, and also dynamic objects, such as instances of classes (objects). Given this, we can use different Object-Z classes to model the different language constructs of an OOPL. For instance, if expressions are modelled by an Object-Z class, say *Expr*, then any individual expression in a program is modelled as an instance of the Object-Z class *Expr*. Similarly, if the construct method is modelled by an Object-Z class, say *Method*, then any individual method in a program is modelled as an instance of the Object-Z class *Method*.

When modelling a language construct, if it needs to be further classified into more specific constructs, then one can use different Object-Z classes to model these constructs. For example, suppose there are two different kinds of statements, such as an assignment statement (like a := exp;) and a method-call (like a.method(...);). Different Object-Z classes, say *AssignStmt* and *MethodCall*, can then model the assignment statement construct and the method-call construct respectively. Any assignment statement in a program is modelled as an instance of the Object-Z class *AssignStmt* and any method-call statement in the program is modelled as an instance of the Object-Z class *MethodCall*. Alternatively, if an entity, say s, refers to a general statement (either an assignment or a method-call) then s can be modelled in Object-Z as an element of the class-union[29] $AssignStmt \cup MethodCall$.

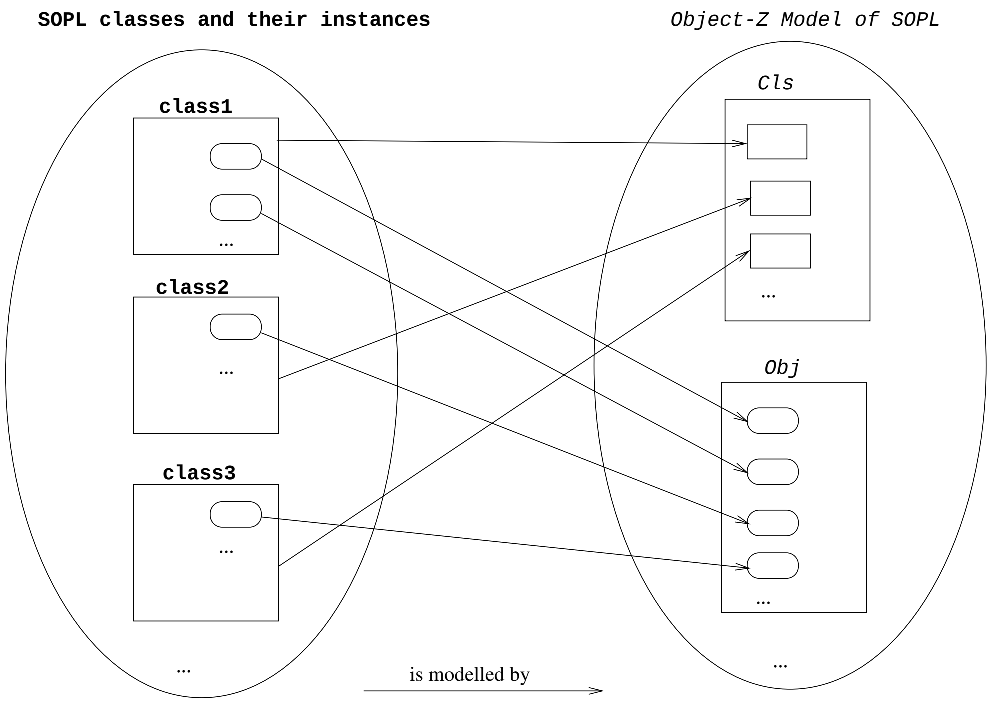


Figure 10.1: Classes and Objects of SOPL and their Object-Z Model.

If the OOPL construct class is modelled by an Object-Z class, say *Cls*, then any individual class in a program is modelled as an instance of the Object-Z class *Cls*. A run-time object in a program can also be modelled as a class, say *Obj*, in Object-Z: that is, each individual object in the program is modelled as an instance of the Object-Z class *Obj*. (Figure 10.1 shows the relationship of SOPL and the Object-Z model of SOPL.) Again, if objects need to be further classified into different categories, say predefined integer objects and those objects which are instantiated from classes defined by a programmer, then different Object-Z classes can be used to model these categories.

Classes and objects are the crucial concepts in object-oriented programming. The relationship between a class and an object in a program is that an object is an instance of a class and a class is a collection of objects. If classes and objects are modelled as Object-Z classes *Cls* and *Obj* respectively, then the relationship between a class and an object can be modelled as cross-references in *Cls* and *Obj*:

Cls

$insts : \mathbb{P}\, Obj_{©}$

...

$\forall\, o : insts \bullet o.type = self$

...

...

Obj

$type : Cls$

...

...

Note that *Cls* and *Obj* are examples in this section only.

In *Cls*, the semantics of the subscript '$_{©}$' appended to the type of the attribute *insts* (instances) implies that:

$$\forall\, c_1, c_2 : Cls \bullet c_1 \neq c_2 \Rightarrow c_1.insts \cap c_2.insts = \varnothing$$

i.e. the object containment notation captures the notion that objects are partitioned into classes.

10.2 An Object-Oriented Model of an OOPL

In this section, a simple object programming language (SOPL) is introduced and its concrete syntax presented. Then the semantics of SOPL is specified in Object-Z. In our model of SOPL we model run-time objects and classes individually, with their relationship modelled by cross-referencing. As the definition of classes requires the understanding of SOPL constructs, such as methods, models of run time objects and variable references (pointers to objects) are first developed, followed by models of expressions, statements, methods, classes and programs.

10.2.1 SOPL and Its Concrete Syntax

The simple object programming language (SOPL) modelled in this chapter is based on the typed basic object programming language BOPL[1][92] with single-inheritance.

[1]The language BOPL contains the main features (classes, objects, assignments and late binding) common to Simula, Smalltalk, C++ and Eiffel.

The rules for assignments and method-calls follow Eiffel's conformance rule. For example, for the assignment $\mathsf{b} := \mathsf{a}$ (where the type of b is $\mathsf{class\ B}$ and the type of a is $\mathsf{class\ A}$) to be valid, either A must be equal to B or A must be a descendant of B. In this section, we suppose SOPL contains only one predefined type, *integer*; in Section 10.3, the language is extended to have also the predefined type *boolean*.

The concrete syntax of SOPL is:

$$
\begin{array}{rcl}
\mathit{Program} & \rightarrow & \mathit{Classes}\ \text{‘Program’}\ \mathit{Id}\ \mathit{Declist}\ \mathit{SeqStmt} \\
\mathit{Classes} & \rightarrow & \epsilon \mid \mathit{Class}\ \mathit{Classes} \\
\mathit{Class} & \rightarrow & \text{‘Class’}\ \mathit{Id}\ \ [\text{‘inherit’}\ \mathit{Id}\ ]\ \mathit{Declist}\ \mathit{Methods} \\
\mathit{Id} & \rightarrow & \mathit{Letter} \mid \mathit{Letter}\ \mathit{Id} \\
\mathit{Letter} & \rightarrow & \text{‘a’} \mid \text{‘b’} \mid \text{‘c’} \mid \ldots\ \text{‘z’} \\
\mathit{Declist} & \rightarrow & \epsilon \mid \mathit{Declaration}\ \mathit{Declist} \\
\mathit{Declaration} & \rightarrow & \mathit{Var}\ \text{‘:’}\ \mathit{Type}\ \text{‘;’} \\
\mathit{Methods} & \rightarrow & \epsilon \mid \mathit{Method}\ \mathit{Methods} \\
\mathit{Method} & \rightarrow & \text{‘Method’}\ \mathit{Id}\ \text{‘(’}\ \mathit{Declist}\ \text{‘);’}\ \mathit{SeqStmt} \\
\mathit{Var} & \rightarrow & \mathit{Id} \\
\mathit{Type} & \rightarrow & \mathit{Id} \mid \text{‘Int’} \\
\mathit{SeqStmt} & \rightarrow & \text{‘begin’}\ \mathit{Stmtseq}\ \text{‘end’} \\
\mathit{Stmtseq} & \rightarrow & \epsilon \mid \mathit{Stmt}\ \mathit{Stmtseq} \\
\mathit{Stmt} & \rightarrow & \mathit{AssignStmt} \mid \mathit{MethodCall} \mid \mathit{CreateStmt} \mid \mathit{ReleaseStmt} \\
\mathit{AssignStmt} & \rightarrow & \mathit{Var}\ \text{‘:=’}\ \mathit{Exp}\ \text{‘;’} \\
\mathit{Exp} & \rightarrow & \mathit{IntExp} \mid \mathit{Var} \mid \mathit{ComponentExp} \\
\mathit{IntExp} & \rightarrow & \mathit{IntConst} \mid \mathit{IntExp}\ \text{‘+’}\ \mathit{IntExp} \\
\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’} \\
\mathit{ComponentExp} & \rightarrow & \mathit{Id}\ \text{‘.’}\ \mathit{Id} \mid \mathit{Id}\ \text{‘.’}\ \mathit{ComponentExp} \\
\mathit{MethodCall} & \rightarrow & \mathit{Id}\ \text{‘.’}\ \mathit{Id}\ \text{‘(’}\ \mathit{Explist}\ \text{‘);’} \\
\mathit{Explist} & \rightarrow & \epsilon \mid \mathit{Exp}\ \text{‘,’}\ \mathit{Explist} \\
\mathit{CreateStmt} & \rightarrow & \text{‘Create(’}\ \mathit{Id}\ \text{‘);’} \\
\mathit{ReleaseStmt} & \rightarrow & \text{‘Release(’}\ \mathit{Id}\ \text{‘);’}
\end{array}
$$

where ϵ denotes the empty string. Here is an example of a small SOPL program.

```
Class Point
x: Int; y: Int;
Method SetToOne();
begin
x := 1; y := 1;
end
Method Move(a: Int; b: Int;);
begin
x := x + a;
y := y + b;
end
```

```
Class TwoPoints
p: Point; q: Point;
Method Init();
begin
Create(p); Create(q);
p.SetToOne(); q.SetToOne();
end
Method MoveQbyP();
begin
q.Move(p.x, p.y);
end
```

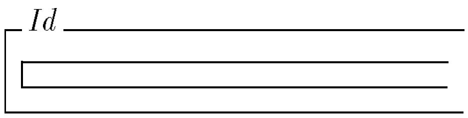
```
Program TP
twopts: TwoPoints;
begin
Create(twopts); twopts.Init; twopts.MoveQbyP();
end
```

In the following sub-sections, the Object-Z model of SOPL is presented.

10.2.2 Modelling Identifiers, Objects and Variable References

Modelling Identifiers

Let the Object-Z class



denote all the possible identifiers (including program names, class names, class attributes and method names) in SOPL. Because we are not interested in the content of an identifier, the class *Id* has no attributes or operations. *Id* is modelled as a class (rather than a given type, such as $[Id]$) so that various Object-Z class modifiers such as object containment can be applied as required.

Modelling Objects

Run-time objects in SOPL are integer objects (predefined) and objects which are instantiated from the user-defined classes. An integer object has an integer value, while an instantiated object has a set of variable references (pointers to objects). Objects of user-defined classes need to be explicitly created by a create-statement. When an object is created, it is assumed that attributes (variable references) of the object are initially unlocated. In this case, the unlocated variable reference points to a special kind of object called the *nil* object. In this model, three Object-Z classes *IntObj*, *InstObj* and *NilObj* define these three different kinds of objects (i.e. integer object, instantiated object, nil object) respectively. Objects in general are modelled as a class-union:

$$pObject \mathrel{\widehat{=}} IntObj \cup InstObj \cup NilObj$$

Let the Object-Z class

NilObj

State	Output
$obj : pObject$	$obj! : pObject$
$obj = self$	$obj! = obj$

model a nil object. A nil object has an attribute *obj* which is a self identity so that the self identity can be broadcast. The class *NilObj* is inherited to define the classes *IntObj* and *InstObj* as will be shown below.

The range of the integer values in SOPL is from -2^{29} to $2^{29}-1$,

$$IntRange == -2^{29}..(2^{29}-1).$$

We view each integer value of SOPL as unique object. Let

IntObj

$NilObj$

$$type : IntClass$$
$$val : IntRange$$
$$\#\{i : IntObj \bullet i.val\} = \#IntRange = \#IntObj$$

model all possible integer objects in SOPL. (The integer class is modelled by *IntClass* which will be defined in Section 10.2.5.) The class invariant of the class *IntObj* specifies that each integer object is uniquely associated with an integer value of *IntRange* and vice-versa.

Let the Object-Z class

$$\begin{array}{|l}
\hline
\textit{InstObj} \\
\textit{NilObj} \\
\hline
\textit{type} : \textit{UsrdefClass} \\
\textit{attris} : \mathbb{P}\ \textit{VarRef}_{©} \\
\hline
\end{array}$$

model instantiated objects from user defined classes and nil objects.

In *IntObj* or *InstObj*, the attribute *type* specifies that the type of an object is the class to which the object belongs. In *IntObj*, the attribute *val* specifies that an integer object has an integer value, while in *InstObj*, the attribute *attris* specifies that an instantiated object has a set of attribute variable references (variable references are modelled by the class *VarRef* which will be defined in the next subsection and the user-defined class is modelled by the *UsrdefClass* which will be defined in Section 10.2.5). The object containment notation attached to the type of the *attris* in *InstObj* specifies that each instantiated object has its own distinct local variable references.

Modelling Variable References

Following the classification of objects, variable references to objects can be divided into variable references to integer objects and variable references to instantiated objects of user-defined classes.

$$\textit{VarRef} \mathrel{\widehat{=}} \textit{IntRef} \cup \textit{InstRef}$$

An integer variable reference and a variable reference of instantiated object have similarities. For example, they all have an identifier, a declared type (class) and a pointer to an object, and their common operations enable the reference to be reattached to another object and to output the object to which it refers. Let

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{CommonRef} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\begin{array}{|l}
\hline
id : Id \\
type : \mathbb{P}Class \\
obj : \mathbb{P}Object \\
\hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \textsc{Init} \_\_\_\_\_\_\_\_} \\
obj \in NilObj \\
\hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ Reattach \_\_\_\_\_\_\_\_} \\
\Delta(obj) \\
obj? : \mathbb{P}Object \\
\hline
obj' = obj? \\
\hline
\end{array} \\
\\
Output \mathrel{\widehat=} obj.Output \\
\hline
\end{array}
$$

model the common structure of these variable references. It is assumed that every variable reference points initially to a nil object.

The integer variable references and instantiated variable references can be modelled as:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ IntRef \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
CommonRef \\
\begin{array}{|l}
\hline
type \in IntClass \\
obj \notin NilObj \Rightarrow \\
\quad obj.type = type \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ InstRef \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
CommonRef \\
\begin{array}{|l}
\hline
type \in UsrdefClass \\
obj \notin NilObj \Rightarrow \\
\quad (type = obj.type \lor \\
\quad type \in obj.type.scontain) \\
\hline
\end{array} \\
\hline
\end{array}
$$

The class invariants of *IntRef* and *InstRef* specify that a variable reference can point to an object whose class is the declared type of the reference. Furthermore, the invariant of *InstRef* specifies polymorphism by allowing the variable reference to point to an object of any derived class of the declared class. The term $obj.type.scontain$ specifies the collection of all the parent (inheritance) class of the class specified by $obj.type$. (See Section 10.2.5 for the specification of classes. A detailed description of the semantics of the attribute *scontain* can be found in Chapter 7.)

10.2.3 Modelling Expressions

The meaning of an expression denotes an object, while the meaning of a statement is to let variable references reattach to other objects. An expression in SOPL can be an instantiated variable reference (e.g. twopts), an integer expression (e.g. x + a) or a component expression (e.g. twopts.p). In Object-Z it is modelled as a class-union

$$Exp \mathrel{\widehat{=}} InstRef \cup IntExp \cup ComponentExp$$

An integer expression can be an integer object (constant) or an integer variable reference or a plus expression or a component integer expression (e.g. 'twopts.p.x' where the type of x is integer), i.e.

$$IntExp \mathrel{\widehat{=}} IntObj \cup IntRef \cup PlusExp \cup IntComponentExp$$

Modelling Plus Expressions

Plus expressions can be modelled as a class:

PlusExp

$e_1, e_2 : IntExp_{\circledS}$
$type : IntClass$
Δ
$obj : IntObj$

$obj.val = (e_1.obj.val + e_2.obj.val)$

Output

$obj! : pObject$

$obj! = obj$

The subscript '$\circledS$' (sharable containment) appended to the type of e_1 and e_2 specifies that the object reference structure of a plus-expression is acyclic (e.g. an expression cannot be a sub-expression directly or indirectly of itself). The type of a plus expression is the integer class which is specified by the attribute *type*. The meaning (dynamic semantics) of a plus expression is an integer object whose value depends on the sub-components of the plus expression; this meaning is specified by the secondary attribute *obj*. A feature of a plus expression is to output its meaning which is specified by the operation *Output*.

The component integer expression will be modelled after the following general component expression is specified.

Modelling Component Expressions

Syntactically, a component expression is a list of identifiers with dots as the separation marks. Abstractly, a component expression is a list of variable references with a validity constraint (static semantic condition). For example, suppose twopts.p.x is a component expression, then p must be an attribute variable of the object referenced by twopts and x must be an attribute of the object referenced by twopts.p. The type of a component expression is the type of the last variable reference in the list. The meaning of a component expression denotes the object referenced by the last variable reference in the list. The following is a model of component expressions:

$$
\begin{array}{|l}
\hline
\textit{ComponentExp} \\
\hline
\textit{first} : \textit{VarRef}_{Ⓢ} \\
\textit{list} : \operatorname{seq} \textit{Id} \\
\Delta \\
\textit{type} : \mathbb{P}\,\textit{Class} \\
\textit{obj} : \mathbb{P}\,\textit{Object} \\
\hline
\# \textit{list} > 1 \\
\exists\, s : \operatorname{seq} \textit{VarRef} \bullet \# s = \# \textit{list} \wedge \{ i : \operatorname{dom} s \bullet (i, s(i).\textit{id}) \} = \textit{list} \\
\quad \forall\, i : 1..(\# s - 1) \bullet s(i).\textit{obj} \in \textit{InstObj} \wedge s(i+1) \in s(i).\textit{obj}.\textit{attris} \\
\quad \textit{first} = \textit{head}(s) \wedge \textit{type} = \textit{last}(s).\textit{type} \wedge \textit{obj} = \textit{last}(s).\textit{obj} \\
\hline
\textit{Output} \mathrel{\widehat{=}} [\textit{obj}! : \mathbb{P}\,\textit{Object} \mid \textit{obj}! = \textit{obj}] \\
\hline
\end{array}
$$

The class invariant specifies the dependency relationship between the first variable reference and the object to which the expression refers.

Now a component integer expression is just a particular kind of component expression.

$$
\begin{array}{|l}
\hline
\textit{IntComponentExp} \\
\hline
\textit{ComponentExp} \\
\hline
\textit{obj} \in \textit{IntObj} \\
\hline
\end{array}
$$

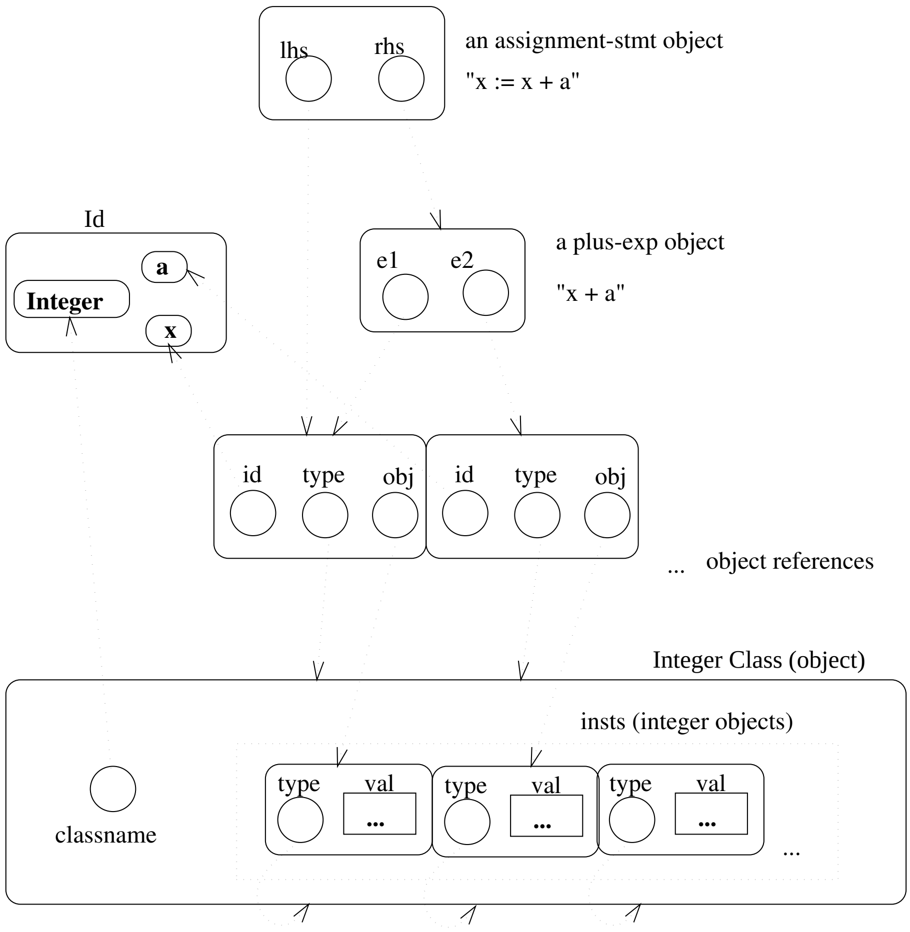


Figure 10.2: Object Reference Structure of an *Assignment* Statement.

10.2.4 Modelling Statements

In SOPL, a statement can be an assignment statement, a create-statement, a release-statement or a method-call statement.

$$Stmt \mathrel{\widehat{=}} AssignStmt \cup CreateStmt \cup ReleaseStmt \cup MethodCall$$

Modelling Assignment Statements

An assignment statement consists of a left hand side variable reference and a right hand side expression. It is modelled as:

$$
\begin{array}{|l}
\textit{AssignStmt} \\
\hline
\begin{array}{|l}
\hline
lhs : VarRef_{Ⓢ} \\
rhs : Exp_{Ⓢ} \\
\hline
lhs.type = rhs.type \lor lhs.type \in rhs.type.scontain \\
\hline
\end{array} \\
\\
Execute \mathrel{\widehat=} rhs.Output \parallel lhs.Reattach \\
\hline
\end{array}
$$

The class invariant specifies that the variable reference (left hand side of an assignment) can reattach to an object of the declared class of the reference or any object of a descendent class of the declared class.

In the definition of the class *AssignStmt*, the abstract syntax of an assignment statement is represented by state attributes *lhs* (left hand side of the assignment) and *rhs* (right hand side of the assignment). The static semantics of the assignment statement is captured by the state invariant. The dynamic semantics of the assignment statement is modelled by the operation *Execute* which relocates the variable reference l to point to the object which is denoted by the expression r. The parallel operator '$\|$' conjoins constraints and equates variables with the same name and also equates inputs and outputs with the same basename. The object reference structure of an *assignment* statement is illustrated by Figure 10.2.

Modelling Create and Release Statements

The meaning of a create-statement on a reference, say Create(p) (where p: Point), is to let the declared class Point make a new Point object and to let the reference p point to the new object. In contrast, the meaning of a release-statement on a reference, say Release(p) (where p: Point), is to let the reference p point to a nil object. The statements to create a new object or to release an object from a reference can be modelled by the following classes.

CreateStmt

$$r : InstRef$$

$$Execute \mathrel{\widehat=} r.type.Create \parallel r.Reattach$$

ReleaseStmt

$$r : InstRef$$

$$Execute \mathrel{\widehat=} [n : NilObj] \bullet n.Output \parallel r.Reattach$$

In the *CreateStmt* definition of the operation *Execute*, *r.type* refers to a user-defined class (the declared type of r). *Create* is an operation of the *UsrdefClass* which will be defined in Section 10.2.5.

Modelling a Sequence of Statements

Before specifying a method-call, we model a sequence of statements. A sequence of statements consists of a list of statements. Its meaning is to execute the statements in the sequence one by one. It is modelled the same as in Section 9.2.2:

SeqStmt

$$stmts : \operatorname{seq} Stmt_{Ⓢ}$$

$$Execute \mathrel{\widehat=} \mathbin{;}\ i : 1..\#stmts \bullet stmts(i).Execute$$

which again demonstrates the reusability of this approach.

Modelling Methods

To define a method-call statement, first we need to model a method of a class. A method is an operation which is defined in a class. The internal state of an object of a class can be changed by the execution of a method of the class. A method consists of a name, a formal parameters list (*fpl*), a set of local variables and a body statements.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Method}} \\
\hline
\begin{array}{|l}
\hline
id : Id \\
fpl : \mathrm{seq}\ VarRef_{Ⓢ} \\
locvars : \mathbb{P}\ VarRef_{Ⓢ} \\
body : SeqStmt_{Ⓢ} \\
\hline
\#\{r : (VarRef \cap scontain) \bullet r.id\} = \#(VarRef \cap scontain) \\
\hline
\end{array} \\
Execute \mathrel{\widehat{=}} body.Execute \\
\hline
\end{array}
$$

The class invariant specifies that, in a method, any two variable references having the same identifier must be the same variable reference. The meaning of a method is the execution of the body statements of the method ($body.Execute$). Notice that there is no '©' appended on the type of the attribute *id* so that two distinct methods in two different classes are allowed to have the same name (unlike in the case of class names).

Modelling Method-call Statements

A method-call statement in SOPL is like a procedure call in Pascal. It consists of a receiver (variable reference), a method of the declared class of the receiver and an actual parameter list (*apl*).

MethodCall

$$
\begin{array}{ll}
recv : InstRef_{\circledS} & \\
m : Method & \\
apl : \mathrm{seq}\, Exp_{\circledS} & \\
\Delta & \\
ps : VarRef \rightarrowtail VarRef & \\
\hline
ps = \{(r_1, r_2) : VarRef \times VarRef \mid r_1 \in recv.obj.attris \land & \\
\qquad r_2 \in m.body.scontain \land r_1.id = r_2.id\} & \\
m \in recv.type.scontain & [\mathsf{inv1}] \\
\# apl = \# m.fpl & [\mathsf{inv2}] \\
\forall\, i : \mathrm{dom}\, apl \bullet m.fpl(i).type = apl(i).type & [\mathsf{inv3}] \\
\qquad \lor m.fpl(i).type \in apl(i).type.scontain &
\end{array}
$$

$$PmtSub \mathrel{\widehat=} \bigwedge i : \mathrm{dom}\, apl \bullet apl(i).Output \parallel m.fpl(i).Reattach$$

$$RelocM \mathrel{\widehat=} \bigwedge (r_1, r_2) : ps \bullet r_1.Output \parallel r_2.Reattach$$

$$RelocR \mathrel{\widehat=} \bigwedge (r_1, r_2) : ps \bullet r_2.Output \parallel r_1.Reattach$$

$$Execute \mathrel{\widehat=} [recv.obj \notin NilObj] \bullet\ PmtSub \mathbin{⨾} RelocM \mathbin{⨾} m.Execute \mathbin{⨾} RelocR$$

The dependent attribute ps pairs variable references of the same name which both appeared as part of the receiver object's state and are used in the calling method's body statements.

The class invariant inv1 specifies that the calling method is defined in the declared class of the receiver (variable reference). The class invariant inv2 specifies that the number of the actual parameters from a method-call statement is equal to the number of formal parameters in the calling method. The class invariant inv3 specifies the conformance rule of parameter substitutions in a method-call.

The operation $PmtSub$ specifies the substitution of the actual (expression) parameters of the method-call to the formal (variable reference) parameters of the method.

The operation $RelocM$ specifies the relocation of the variable references occurring in the method body to point to the objects referenced by the attribute variables of the receiver object. The operation $RelocR$ specifies the reverse of the process captured by the operation $RelocM$. With these process relocations, the execution of the method ($m.Execute$) can effectively update the state of the object.

10.2.5 Modelling Classes

A class is a template for objects: each object of the class has a state which conforms to the signature of the class and the state of the object can be changed by invoking a method of the class.

In object-oriented programming languages, classes are often used as types. There are two kinds of classes, predefined classes (e.g. integer class) and user-defined classes defined by the programmer. Let

$$
\begin{array}{|l}
\textit{IntClass} \\
\hline
\textit{classname} : \textit{Id}_{©} \\
\textit{insts} : \mathbb{P}\,\textit{IntObj}_{©} \\
\hline
\#\textit{IntClass} = 1 \land \textit{insts} = \textit{IntObj}
\end{array}
$$

model the integer class of the language. In the class invariant part of $IntClass$, the predicate $\#IntClass = 1$ specifies that there is only one integer class in SOPL. The second conjunct specifies that all possible integer objects are contained by the integer class.

From the inheritance point of view, there are two kinds of user-defined classes — root classes (classes defined without inheriting another class) and derived classes (classes defined by inheriting another class). In Object-Z, the user-defined classes in a program can be modelled as a class-union:

$$UsrdefClass \mathrel{\widehat=} RootClass \cup DerivedClass$$

where *RootClass* and *DerivClass* will be defined later. Similarly, SOPL classes can also be defined as a class-union:

$$pClass \mathrel{\widehat=} IntClass \cup UsrdefClass$$

As a root class and a derived class are quite similar (they all have a class name, a signature, a set of methods and a set of created instances), we can model their common part as the Object-Z class *CommonClass*, and then the *RootClass* and *DerivedClass* can be defined by inheriting *CommonClass*.

CommonClass

$$\begin{array}{l}
classname : Id_{\copyright} \\
sig : Id \mathrel{+\!\!\!\!\to} pClass \\
methods : \mathbb{P}\, Method_{\copyright} \\
insts : \mathbb{P}\, InstObj_{\copyright} \\
\Delta \\
totalsig : Id \mathrel{+\!\!\!\!\to} pClass \\
totalms : \mathbb{P}\, Method \\
\hline
\forall\, o : insts \bullet o.type = self \\
\forall\, o : insts \bullet \{r : o.attris \bullet (r.id, r.type)\} = totalsig \\
\#totalms = \#\{m : totalms \bullet m.id\} \\
\forall\, m : totalms \bullet \{r : VarRef \mid \\
\quad r \in m.scontain \setminus \operatorname{ran} m.fpl \setminus m.locvars \bullet (r.id, r.type)\} \subseteq totalsig
\end{array}$$

Create

$$\begin{array}{l}
\Delta(insts) \\
obj! : pObject \\
\hline
\exists\, o : InstObj \bullet \forall\, r : o.attris \bullet r.\textsc{Init} \\
\qquad\qquad o \notin insts \land insts' = insts \cup \{o\} \land obj! = o
\end{array}$$

The object-oriented view of the example class Point (given in Section 10.2.1) can be illustrated in a graphical form in Figure 10.3.

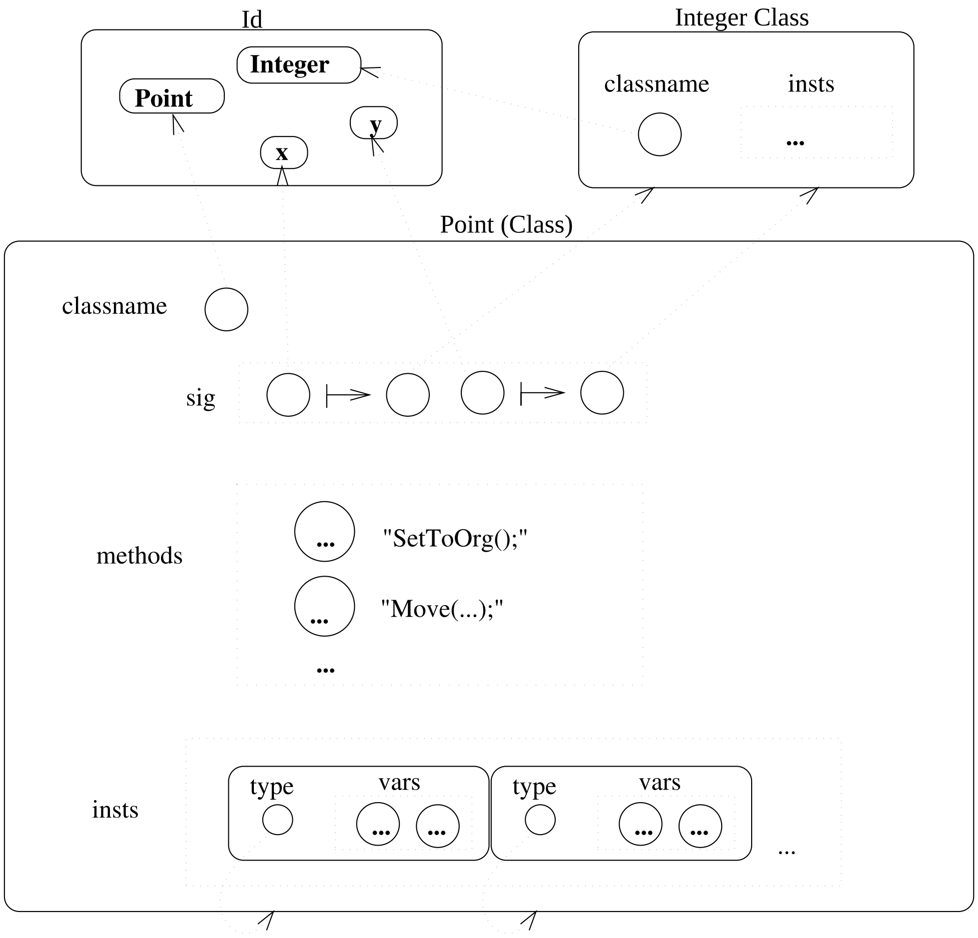


Figure 10.3: Object Reference Structure of the 'Point' Class.

In *CommonClass*, *sig* denotes the signature of the attributes explicitly declared in a class, while the dependent attribute *totalsig* denotes the total signature of the variables explicitly declared in the class and implicitly inherited; *methods* denotes the methods explicitly defined in the class, while *totalms* denotes all the methods explicitly defined in the class and implicitly inherited (the precise meaning of *totalsig* and *totalms* is defined in the class invariants of *RootClass* and *DerivClass*); *insts* denotes all the created objects of the class.

The subscript '$_{\copyright}$' appended on the type of *classname* specifies that any two different classes have different names. The subscript '$_{\copyright}$' appended on the type of *methods* indicates that a class contains a distinct set of (locally) explicitly defined methods.

The subscript '$_{\copyright}$' appended on the type of *insts* specifies that objects are partitioned into classes.

The class invariants of *CommonClass* specify that

- all objects contained by a class must belong to the class;
- every object must have the same signature of its class;
- every method of a class has a distinct method name; (note that it still allows two different classes to have two different methods with the same name);
- variable references (except formal parameters and local variables) accessed in a method must be declared in the class. Note that the term $m.scontain$ denotes the objects which are directly and indirectly contained (but sharable) by the object referenced by m.

The operation *Create* specifies that a new object can be created by its class and the address (identity) of the object can be output to the environment. Although we are not concerned with the way in which a new object is created, by adding this operation in *CommonClass* we capture the notion that a class is a factory for its objects. Also, note that the operation *Create* is only defined for modelling a user-defined class, whereas in the case of modelling the integer class, it is not necessary to define such an operation because all integer objects are predefined and do not need to be explicitly created by a create-statement.

Now, *RootClass* and *DerivedClass* can be defined by inheriting *CommonClass*:

RootClass

$CommonClass$

$$totalsig = sig$$
$$totalms = methods$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DerivedClass}} \\
\hline
\textit{CommonClass} \\
\quad \begin{array}{|l}
\hline
\textit{parclass} : \textit{UsrdefClass}_{\circledS} \\
\hline
\textit{totalsig} = \textit{sig} \cup \textit{parclass.totalsig} \\
\textit{totalms} = \textit{methods} \cup \textit{parclass.totalms} \\
\hline
\end{array} \\
\hline
\end{array}$$

In *DerivedClass*, *parclass* denotes the parent class of a derived class. The subscript '$\circledS$' (sharable containment) appended on the type of *parclass* precisely specifies that the single inheritance structure is acyclic (i.e. a class cannot directly or indirectly inherit itself). The attribute *methods* (inherited from *CommonClass*) denotes the locally defined additional methods.

10.2.6 Programs

A program is similar to a method of a class. It consists of a set of declared variable references and a sequence of statements to manipulate them.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Program}} \\
\hline
\quad \begin{array}{|l}
\hline
\textit{vars} : \mathbb{P}\ \textit{VarRef} \\
\textit{body} : \textit{SeqStmt} \\
\hline
\forall r_1, r_2 : \textit{vars} \bullet r_1 \neq r_2 \Rightarrow r_1.\textit{id} \neq r_2.\textit{id} \\
\textit{body.scontain} \cap \textit{VarRef} = \textit{vars} \\
\hline
\end{array} \\
\\
\quad \begin{array}{|l}
\multicolumn{1}{l}{\textsc{Init}} \\
\hline
\forall r : \textit{vars} \bullet r.\textsc{Init} \\
\hline
\end{array} \\
\\
\textit{Execute} \mathrel{\widehat=} \textit{body.Execute} \\
\hline
\end{array}$$

The first class invariant specifies that any two declared variables have different names in a program. The second class invariant specifies that every occurrence of variable referenced in the body of the program must be declared in *vars*.

10.3 The Ease of Extendibility

One aim of using an object-oriented approach in the language model is to achieve the extendibility of the language model.

10.3.1 Adding More Language Constructs

Suppose that SOPL is extended to have the following language constructs:

- predefined *Boolean* class, predefined *Boolean* objects and *Boolean* variable references;
- less-than expression;
- while statement.

Then, in our model it is necessary only to:

(1) add the following classes:

$$
\begin{array}{|l}
\textit{BoolObj} \\
\hline
\textit{NilObj} \\
\begin{array}{|l}
\hline
\textit{type} : \textit{BoolClass} \\
\textit{val} : \mathbb{B} \\
\hline
\#\{b : \textit{BoolObj} \bullet b.\textit{val}\} = \#\mathbb{B} = \#\textit{BoolObj} \\
\hline
\end{array} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{BoolClass} \\
\hline
\begin{array}{|l}
\hline
\textit{classname} : \textit{Id}_{©} \\
\textit{insts} : \mathbb{P}\, \textit{BoolObj}_{©} \\
\hline
\#\textit{BoolClass} = 1 \\
\textit{insts} = \textit{BoolObj} \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{BoolRef} \\
\hline
\textit{CommonRef} \\
\begin{array}{|l}
\hline
\textit{id} : \textit{Id} \\
\textit{type} : \textit{BoolClass} \\
\hline
\textit{obj}.\textit{type} = \textit{type} \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{BoolComponentExp} \\
\hline
\textit{ComponentExp} \\
\begin{array}{|l}
\hline
\textit{obj} \in \textit{BoolObj} \\
\hline
\end{array} \\
\hline
\end{array}
$$

LessExp

$$
\begin{array}{l}
e_1, e_2 : IntExp_{Ⓢ} \\
type : BoolClass \\
\Delta \\
obj : BoolObj \\
\hline
obj.val = (e_1.obj.val < e_2.obj.val)
\end{array}
$$

Output

$$
\begin{array}{l}
obj! : pObject \\
\hline
obj! = obj
\end{array}
$$

WhileStmt

$$
\begin{array}{l}
test : BoolExp_{Ⓢ} \\
body : SeqStmt_{Ⓢ}
\end{array}
$$

$$
\begin{array}{rl}
Execute \mathrel{\widehat=} & ([test.obj.val] \land body.Execute) \mathbin{;} Execute \\
& [] \\
& [\neg\ test.obj.val]
\end{array}
$$

(2) extend the class-unions *pClass*, *pObject* and *VarRef* to include the classes *BoolClass*, *BoolObj* and *BoolRef* respectively; add a new class-union

$$BoolExp \mathrel{\widehat=} BoolObj \cup BoolRef \cup LessExp \cup BoolComponentExp$$

to the model; then extend the class-unions *Exp* and *Stmt* to include the classes *BoolExp* and *WhileStmt* respectively.

All classes defined previously in Section 10.2 remain unchanged.

10.3.2 Method Redefinition in Class Inheritance

Now suppose we allow method-redefinition in class inheritance; then the *DerivedClass* can be defined as:

$$
\begin{array}{l}
\textit{DerivedClass} \\
\hline
\textit{CommonClass} \\
\hline
\textit{parclass} : \textit{UsrdefClass}_{\circledS} \\
\Delta \\
\textit{oldms}, \textit{newms} : \mathbb{P}\,\textit{Method} \\
\hline
\textit{oldms} = \{ m_1 : \textit{parclass.totalms} \mid \exists\, m_2 : \textit{methods} \bullet m_1.\textit{id} = m_2.\textit{id} \} \\
\textit{newms} = \{ m_1 : \textit{methods} \mid \exists\, m_2 : \textit{parclass.totalms} \bullet m_1.\textit{id} = m_2.\textit{id} \} \\
\textit{totalsig} = \textit{sig} \cup \textit{parclass.totalsig} \\
\textit{totalms} = \textit{methods} \cup \textit{parclass.totalms} - \textit{oldms} \\
\forall\, m_1 : \textit{oldms}, m_2 : \textit{newms} \bullet m_1.\textit{id} = m_2.\textit{id} \Rightarrow \\
\qquad\qquad \# m_1.\textit{fpl} = \# m_2.\textit{fpl} \wedge \forall\, i : \operatorname{dom} m_2.\textit{fpl} \bullet \\
\qquad\qquad\qquad m_2.\textit{fpl}(i).\textit{id} = m_1.\textit{fpl}(i).\textit{id} \\
\qquad\qquad\qquad (m_2.\textit{fpl}(i).\textit{type} = m_1.\textit{fpl}(i).\textit{type} \\
\qquad\qquad\qquad\quad \vee\, m_2.\textit{fpl}(i).\textit{type} \in m_1.\textit{fpl}(i).\textit{type.scontain})
\end{array}
$$

The attribute *methods* (inherited from *CommonClass*) specifies the additional methods and redefined methods of a derived class. The dependent attribute *newms* specifies a set of redefined methods and *oldms* specifies a set of overwritten methods of the parent class (the precise meaning of *newms* and *oldms* is defined by the first two conjunctions in the class invariants of the *DerivClass*). The last conjunction of the class invariant specifies the conformance rule of method redefinition.

10.4 Conclusion

In this chapter, we have used our object-oriented approach to the specification of the denotational semantics of a simple object-oriented programming language. The key idea of this approach is to model the language constructs of OOPLs using an object-oriented view. The advantages of applying such an approach to model a programming language is that the abstract syntax and static and dynamic semantics for each component of a programming language structure are grouped together and captured in a class construct; hence the language model is concise and easily extended.

As the core features of OOPLs have been modelled in this chapter, we believe that it is feasible to extend this model to capture fully the denotational semantics of commercial object-oriented programming languages, such as Eiffel and C++.

Various object-oriented concepts in object-oriented programming are precisely captured by the various features (e.g. the object containment notion) of the Object-Z notation. For instance, the acyclic structure of class inheritance relationship is precisely specified by the object containment notation '$\circledS$'. Furthermore, this semantic model of SOPL captures various object-oriented concepts. For example, when a class is modelled, the specification precisely captures the combined ideas that a class is a template, a collection and a factory of objects. This chapter in effect does more than define the semantics of OOPLs; it also presents an object-oriented approach to the formalisation of object-oriented concepts themselves.

Chapter 11

Conclusions and Directions for Further Research

This chapter summarises the main contributions of this thesis and discusses possible directions for further research.

11.1 Thesis Main Contributions and Influence

- This thesis successfully applied an object-oriented approach to specify the denotational semantics of programming languages. With this approach, the abstract syntax, static semantics and dynamic semantics of an individual language construct are typically defined in one class such that the semantic representation is structural. Compared to the traditional approach, this object-oriented semantic representation is extendible and reusable. Assuming an the understanding of object-oriented concepts and Object-Z constructs, the object-oriented semantics are much simpler. As object-orientation is becoming more popular and acceptable, reading object-oriented semantics will become easier.

  Although object-oriented ideas applied to attribute grammars[75] and compilation had been discussed in [9, 53, 52, 56, 61, 84, 117], they had focused on language syntax, grammar and compilation issues but hardly on issues related to denotational semantics. We believe that this thesis is the first to demonstrate the feasibility of using object orientation to define the denotational semantics of programming languages. Butler[15] applied this research to implement an interpreter for a procedural language from its formal object-oriented specification.

- This thesis developed new and useful constructs, such as class-union, object containment and exclusive object control, for the Object-Z specification language. These new extensions to Object-Z not only play important roles in the object-oriented definition of programming languages but also prove to be useful for modelling object-oriented systems in general. Also these new extensions to Object-Z have influenced other research activities. For instance, Chen[19] based his notion of class types on the idea of class-union; and Hussey[68] applied the object containment notation to specify the Model-View-Controller (MVC)[49] and Presentation-Abstract-Control (PAC)[24] architectures. Dong and Lin[38] applied the the containment notation to specify an interactive visualisation system. Atkinson[5] used the containment notations when formalising the Eiffel Library Standard[87].

- This thesis detailed the notion of secondary attributes, their roles and implications in formal object-oriented specification. This thesis also compared and discussed the free type and class-union constructs through various examples, and consequently, presented guidelines on how to appropriately and effectively use these constructs in formal object-oriented specification.

11.2 Directions for Further Research

The following topics, arising out of this thesis, seem worthy of further research.

11.2.1 A Standard Object-Z Generic Class Library

In Chapter 2, a number of generic classes are used to illustrate the Object-Z language. These generic classes capture some common object structures and some of them were used as a class library to specify the Open Distributed Processing (ODP) Trader in [30]. The resultant specification of the Trader is found to be concise and extendible.

We believe that a standard Object-Z class library needs to be constructed so that the specification of a specific system can be developed from the standard library. Two areas need to be investigated in order to build a general and reusable class library. The first area is to investigate what are the common abstract class internal structures among the existing Object-Z case studies. The second area is to investigate the existing class libraries of object-oriented programming languages, such as Smalltalk and Eiffel, so that a smooth refinement process can be carried out from specifications to code.

11.2.2 Applying Secondary Attributes to Specify Distributed Systems

The *GameSharingGroup* case study in Chapter 3 demonstrates that the notions of information distribution and information sharing are facilitated by the use of secondary attributes. We believe that secondary attributes will contribute significantly to the formal modelling of distributed systems. Therefore it would be interesting for a further study to apply secondary attributes to specify some real distributed system.

11.2.3 Calculating Polymorphic Cores: Type Checking

Like Z, Object-Z is a strongly typed language and a type checker for Object-Z needs to be developed. One of the challenging tasks for developing a type checker for Object-Z is to type check polymorphic terms and operations. An algorithm for calculating the polymorphic core of a class-union needs to be developed. The algorithm should enable the type narrowing construct to locally change the type environment for the narrowed polymorphic reference.

11.2.4 Converting Containment and Exclusive Control Notations into Predicates

The containment notations ‘$_{\copyright}$’ and ‘$_{\circledS}$’ introduced in Chapter 7 are syntactic conventions. They need to be translated into predicates when one is reasoning about the specifications. For instance, the class

$$
\begin{array}{l}
\textbf{X} \\
\hline
as : \mathbb{P}\, A_{\copyright} \\
bcf : B_{\circledS} \mapsto C_{\copyright} \\
d : D_{\textcircled{E}} \\
es : \mathbb{P}\, E \\
\hline
\end{array}
$$

$\downarrow$

needs to be translated into $\longrightarrow$

$$
\begin{array}{l}
\textbf{X} \\
\nmid (d) \\
\hline
as : \mathbb{P}\, A \\
bcf : B \mapsto C \\
d : D \\
es : \mathbb{P}\, E \\
\hline
dcontain = as \cup \operatorname{ran} bl \cup \operatorname{ran} bcf \\
sdcontain = \operatorname{dom} bcf \\
excon = \{d\} \\
freerefs = es \\
\hline
\end{array}
$$

Therefore, a pre-processor tool is needed to perform such task. A task for further research is to develop such a tool.

11.2.5 Guidelines for Applying Containment and Control Notations

The notions of object containment and exclusive control add various degrees of encapsulation to the object client-supplier associations. In this thesis, a number of examples are presented using the containment and exclusive control constructs. These examples are given mainly to motivate the development of these constructs. Further research is needed to provide some guidelines on how to appropriately and effectively apply these constructs to specify object-oriented systems. For instance, if an object is contained within another and exclusively controlled by that object, then '$_{\textcircled{E}}$' should be used, as the property of exclusive control implies the geometric properties of the containment. To generate appropriate rules for such situations, a number of case studies need to be considered so that the general guidelines can be formulated.

11.2.6 Specifying Safety and Security Critical Systems

The notion of exclusive object control is an important aspect of safety and security critical systems. With the specific notation defined in Chapter 8 of this thesis, Object-Z is a suitable language to specify safety and security critical systems. A task for future research is to apply Object-Z to specify some real safety and security critical systems. Another task for future research is to investigate how the notation for exclusive object control can facilitate reasoning about systems specified in Object-Z.

11.2.7 Semantics of Other Programming Language Paradigms

We have also successfully applied an object-oriented approach to specify the denotational semantics of a block-structured sequential programming language (Chapter 9) and the denotational semantics of an object-oriented programming language (Chapter 10). Whether the approach is suitable for modelling the semantics of other programming language paradigms, such as declarative and parallel programming languages, remains to be explored. Therefore a further research area is to apply the object-oriented approach to specify other programming language paradigms.

Bibliography

[1] A. Alencar and J. Goguen. OOZE: An object oriented Z environment. In P. America, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lect. Notes in Comput. Sci.*, pages 180–199. Springer-Verlag, 1991.

[2] P. America, J. de Bakker, J. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2):152–205, 1989.

[3] D. Andrews and W. Henhapl. Pascal. In D. Bjørner and C. B. Jones, editors, *Formal Specification and Software Development*, International Series in Computer Science, pages 175–252. Prentice-Hall, 1982.

[4] 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.

[5] S. Atkinson. Formalizing the Proposed Eiffel Library Kernel Standard. Technical Report 95-35, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, September 1995. To appear in *Proc. Technology of Object-Oriented Languages and Systems: TOOLS 18*, Prentice Hall 1995.

[6] P. Bancroft and I. Hayes. A Formal Semantics for a Language with Type Extension. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 299–314. Springer-Verlag, September 1995.

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

[8] D. Bjørner and O. N. Oest. *Towards a Formal Desciption of Ada.* Springer-Verlag (*LNCS* 98), 1980.

[9] A. Bloesch and T. Halpin. An object-oriented approach to extensible parsing. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 345–351, February 1993.

[10] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[11] G. Booch. *Object-Oriented Design with Applications.* Addison-Wesley, 1991.

[12] S. Brien, T. King, J. Nicholls, J. Woodcock, and J. Wordsworth. Z Base Standard — Version 1.0. Technical report, Programming Research Group, Oxford Univ., Nov 1992.

[13] Wayne Brookes and Jadwiga Indulska. "ODP Types and Their Management: an Object-Z Specification". In *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing, ICODP'95.* North-Holland, February 1995.

[14] R. Burstall and J. Goguen. An informal introduction to specifications using Clear. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, International Lecture Series in Computer Science, chapter 4, pages 185–213. Academic Press, 1981.

[15] S. Butler. Generation of an Interpreter From a Specification of its Semantics in Object-Z. Honours Report, Department of Computer Science, The University of Queensland, 1994.

[16] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, December 1985.

[17] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, International Symposium (LNCS 173)*, pages 51–67. Springer-Verlag, 1984.

[18] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. North-Holland, 1990.

[19] J. Cheng. Class Types as Sets of Classes in Object-Oriented Formal Specification Languages. In C. Mingins and B. Meyer, editors, *Proc. 15th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 15*, pages 205–215. Prentice-Hall, November 1994.

[20] F. Civello. Role for composite objects in object-oriented analysis and design. In *Proc. 8th ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'93)*, pages 376–393, 1993.

[21] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, 1991.

[22] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. Technical Report STL-89-17, Hewlett-Packard Labs, 1989.

[23] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 433–443, 1989.

[24] Joelle Coutaz. Architectural Models for Interactive Software. In Stephen Cook, editor, *European Conference on Object-oriented Programming - ECOOP '89*, pages 382–399. Cambridge University Press, 1989.

[25] D. de Champeaux, D. Lea, and P. Faure. *Object-Oriented System Development.* Addison-Wesley, 1993.

[26] A. Diller. *Z: An Introduction to Formal Methods.* John Wiley & Sons, 2nd edition, 1994.

[27] T. Dillon and P.L. Tan. *Object-Oriented Conceptual Modeling.* Prentice-Hall, 1993.

[28] 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.

[29] 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.

[30] J. S. Dong and R. Duke. An Object-Oriented Approach to the Formal Specification of ODP Trader. In J. Meer, B. Mahr, and S. Storp, editors, *Open Distributed Processing, II (ICODP'93)*, pages 341–352, Berlin, 1994. North-Holland.

[31] 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.

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

[33] 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 20)*. Prentice-Hall, 1996. to appear.

[34] 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.

[35] J. S. Dong, R. Duke, and G. Rose. A Small Language Definition in Object-Z. Technical Report 95-21, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1995.

[36] 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, Florida, November 1995. IEEE Computer Society Press.

[37] J.S. Dong, J. Colton, and L. Zucconi. Formal Object Approach to Real-Time Specification. Technical Report 96-18, Division of Information Technology, Commonwealth Scientific and Industrial Research Organisation (CSIRO), February 1996.

[38] J.S. Dong and T. Lin. Formalising an Interactive Visualisation System. To appear in the 3rd Eurographics Workshop on Design, Specification and Verification of Interactive Systems. Namur (Belgium), 1996.

[39] D. Duke. *Object-Oriented Formal Specification.* PhD thesis, University of Queensland, 1991.

[40] D. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lect. Notes in Comput. Sci.*, pages 242–262. Springer-Verlag, 1990.

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

[42] 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.

[43] R. Duke and G. Rose. An Object-Z Specification of a Mobile Phone System. In K. Lano and H. Haughton, editors, *Object Oriented Specification Case Studies*, Object Oriented Series. Prentice-Hall, 1993.

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

[45] R. Duke, G. Rose, and A. Lee. Object-oriented protocol specification. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification, X*, pages 325–338. North-Holland, 1990.

[46] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. To appear in a special issue of *Computer Standards and Interfaces* on Formal Methods and Standards, September 1995.

[47] R.W. Floyd. Assigning Meanings to Programs. In *American Mathematical Society Symposium in Applied Mathematics*, pages 19–31, 1967.

[48] J. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing software specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*, pages 391–420. Addison-Wesley, 1985.

[49] A. Goldberg and D. Robson. *Smalltalk 80: The Language.* Addison-Wesley, 1989.

[50] M. J. C. Gordon. Operational reasoning and denotational semantics. In *Proc. Int. Symp. on Proving and Improving Programs*, 1978.

[51] M. J. C. Gordon. *The Denotational Description of Programming Languages, An Introduction.* Springer-Verlag, 1979.

[52] J.O. Graver. T-gen: a String-To-Object Translator Generator. *Journal of Object-Oriented Programming*, 5(6):35–42, 1992.

[53] J.O. Graver. The Evolution of an Object-oriented Compiler Framework. *Software Practice and Experience*, 22(7):519–535, 1992.

[54] A Griffiths. An Extended Semantic Foundation For Object-Z. Technical Report 95-39, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, 1995.

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

[56] J. Grosch. Object-Oriented Attribute Grammars. In *Proc. 5th Fifth Symposium on Computer and Information Sciences*, pages 807–816, 1990.

[57] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, 2nd edition, 1993.

[58] 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.

[59] I.J. Hayes. VDM and Z: A comparative case study. *Formal Aspects of Computing*, 4(1):76–99, 1992.

[60] I.J. Hayes, C.B.Jones, and J.E.Nicholls. VDM and Z: A comparative case study. *FACS Europe, series I*, 1(1):7–30, 1993.

[61] G. Hedin. An Object-Oriented Notation for Attribute Grammars. In *Proc. European Conf. on Object-Oriented Programming (ECOOP'89)*, pages 329–345, 1989.

[62] B. Henderson-Sellers. *A Book of Object-Oriented Knowledge*. Object Oriented Series. Prentice-Hall, 1992.

[63] M. Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, New York, 1990.

[64] Andreas V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Proc. Theoretical Aspects of Computer Software*, pages 548–568. Springer-Verlag (*LNCS* 526), 1991.

[65] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[66] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[67] J. Hogg. Islands: Aliasing Protection In Object-Oriented Languages. In *Proc. 6th ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91)*, pages 271–285, 1991.

[68] Andrew Hussey and David Carrington. Comparing two user-interface architectures: MVC and PAC. In Sandrine Balbo, editor, *QCHI '95 Symposium*, pages

3–21. Computer Human Interaction Special Interest Group of the Ergonomics Society of Australia, 1995.

[69] ISO/IEC JTC1/SC21/WG7 N743. *Working Document on Topic 9.1 - ODP Trader*, November 1992.

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

[71] 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.

[72] Samuel Kamin. Inheritance in Smalltalk–80: A denotational definition. In *Fifteenth Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[73] H. Kilov. Information Modeling and Object-Z: Specifying Generic Reusable Associations. In O. Etzion and A. Segev, editors, *Proceedings of Next Generation Information Technology and Systems*, pages 182–191, June 1993.

[74] H. Kilov and J. Ross. *Information Modeling: An Object-Oriented Approach.* Object Oriented Series. Prentice-Hall, 1994.

[75] D.E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[76] K. Lano. Z++. In *Proc. 1990 Z User's Meeting*, Oxford, December 1990.

[77] K. Lano and H. Haughton. A Comparative Description of Object-oriented Specification Language. In K. Lano and H. Haughton, editors, *Object Oriented Specification Case Studies*, Object Oriented Series. Prentice-Hall, 1993.

[78] A. Lee. *Formal Specification and Analysis of Intelligent Network Services and their Interactions.* PhD thesis, University of Queensland, 1993.

[79] A. Lee and D. Carrington. Formalising extensions and modifications to telecommunication software. In *Proc. 1st Australian Conf. on Telecommunications Software*, pages 205–210, April 1991.

[80] R.H. Liffers. Inheritance versus Containment. *ACM SIGPLAN Notices*, 28(9):36–38, 1993.

[81] A. MacDonald and D. Carrington. Structuring Z Specifications: Some Choices. In J.P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, number 967 in Lecture Notes in Computer Science, pages 203–223, Limerick, Ireland, September 1995. Springer-Verlag.

[82] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, 3(4):184–195, 1960.

[83] S.L. Meira and A.L.C. Cavalcanti. Modular object-oriented z specifications. In *Proc. 1990 Z User's Meeting*, pages 173–192, December 1991.

[84] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1988.

[85] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice-Hall, 1988.

[86] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[87] B. Meyer. The Eiffel Library Standard (*Vintage 95*). Technical Report TR-EI-48/KL, Interactive Software Engineering, Inc., 225 Storke Rd, Suite 7, Goleta, CA 93117 USA, June 1995. Version 8. Available from ftp://ftp.eiffel.com/pub/nice/library.

[88] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1989.

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

[90] O. Nierstrasz. Composing Active Objects — The Next 700 Concurrent Object-Oriented Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 151–171. MIT Press, 1993.

[91] J. Odell. Managing object complexity, part II: composition. *Journal of Object-Oriented Programming*, 5(6):17–20, 1992.

[92] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[93] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[94] Ben Potter, Jane Sinclair, and David Till. *An introduction to formal specification and Z.* Prentice-Hall, 1991.

[95] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[96] 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.

[97] G. Rose, R. Duke, and I. Hayes. Specifying communications services and protocols. In *Proc. 2nd Australian Software Eng. Conf. (ASWEC'87)*, pages 161–170, Canberra, May 1987.

[98] J. Rumbaugh. Derived information. *Journal of Object-Oriented Programming*, 5(1):57–61, 1992.

[99] A. Smith. On recursive free types in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, pages 3–39, University of York, December 1991.

[100] G. Smith. *An Object-Oriented Approach to Formal Specification.* PhD thesis, University of Queensland, 1992.

[101] 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.

[102] G. Smith. Extending W for Object-Z. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, September 1995.

[103] J.M. Spivey. *Understanding Z: A specification language and its formal semantics*, volume 3 of *Cambridge Tracts in Theoretical Comput. Sci.* Cambridge University Press, UK, 1988.

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

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

[106] P. Stocks, K. Raymond, D. Carrington, and A. Lister. Modelling open distributed systems in Z. *Computer Communications*, 15(2):103–113, 1992.

[107] J. E. Stoy. *Denotational semantics : the Scott-Strachey approach to programming language theory.* Cambridge, Mass. : MIT Press, 1977.

[108] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[109] P.A. Swatman. *Increasing Formality in the Specification of High-Quality Information Systems in a Commercial Context.* PhD thesis, The Curtin University of Technology, 1992.

[110] P.A. Swatman, P.M.C. Swatman, and R. Duke. Electronic data interchange: A high-level formal specification in Object-Z. In *Proc. 6th Australian Software Eng. Conf. (ASWEC'91)*, 1991.

[111] R. D. Tennent. The Denotional Semantics of Programming Languages. *Communications of the ACM*, 8:437–453, 1976.

[112] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proc. European Conf. on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lect. Notes in Comput. Sci.*, pages 55–77. Springer-Verlag, 1988.

[113] A. Wills. Capsules and types in Fresco. In P. America, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lect. Notes in Comput. Sci.*, pages 59–76. Springer-Verlag, 1991.

[114] Mario Wolczko. Semantics of Smalltalk-80. In *Proc. European Conference on Object-Oriented Programming*, pages 108–120. Springer-Verlag (*LNCS* 276), 1987.

[115] J. C. P. Woodcock and S. M. Brien. W: A logic for Z. In J. E. Nicholls, editor, *the Sixth Annual Z User Meeting, York, UK.*, Workshops in Computing, pages 77–96. Springer-Verlag, 1992.

[116] J. B. Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.

[117] P.C. Wu and F.J. Wang. An Object-Oriented Specification for Compilers. *ACM SIGPLAN Notices*, 27(1):85–94, 1992.

Appendix A

Glossary of Z Notation

This appendix presents a glossary of the Z notation. It is based on the glossary of Z notation presented in Hayes[57].

Mathematical Notation

Definitions and declarations

Let x, x_k be identifiers and let T, T_k be non-empty, set-valued expressions.

$LHS == RHS$ — Definition of LHS as syntactically equivalent to RHS.

$LHS[X_1, X_2, \ldots, X_n] == RHS$
Generic definition of LHS, where $X_1, X_2, \ldots, X_n$ are variables denoting formal parameter sets.

$x : T$ — A declaration, $x : T$, introduces a new variable x of type T.

$x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$
List of declarations.

$x_1, x_2, \ldots, x_n : T$ — $== x_1 : T;\ x_2 : T;\ \ldots;\ x_n : T$

$[X_1, X_2, \ldots, X_n]$ — Introduction of free types named $X_1, X_2, \ldots, X_n$.

Logic

Let P, Q be predicates and let D be a declaration or a list of declarations.

true, *false*	Logical constants.
$\neg P$	Negation: "not P".
$P \wedge Q$	Conjunction: "P and Q".
$P \vee Q$	Disjunction: "P or Q or both".
$P \Rightarrow Q$	$== (\neg P) \vee Q$ Implication: "P implies Q" or "if P then Q".
$P \Leftrightarrow Q$	$== (P \Rightarrow Q) \wedge (Q \Rightarrow P)$ Equivalence: "P is logically equivalent to Q".
$\forall x : T \bullet P$	Universal quantification: "for all x of type T, P holds".
$\exists x : T \bullet P$	Existential quantification: "there exists an x of type T such that P holds".
$\exists_1 x : T \bullet P$	Unique existence: "there exists a unique x of type T such that P holds".

$\forall x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$

"For all x_1 of type T_1, x_2 of type T_2, ..., and x_n of type T_n, P holds."

$\exists x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$

Similar to $\forall$.

$\exists_1 x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$

Similar to $\forall$.

$\forall D \mid P \bullet Q$	$\Leftrightarrow \forall D \bullet P \Rightarrow Q$

$\exists D \mid P \bullet Q$	$\Leftrightarrow \exists D \bullet P \land Q$
$t_1 = t_2$	Equality between terms.
$t_1 \neq t_2$	$\Leftrightarrow \neg (t_1 = t_2)$

Sets

Let X be a set; S and T be subsets of X; t, t_k terms; P a predicate; and D declarations.

$t \in S$	Set membership: "t is a member of S".
$t \notin S$	$\Leftrightarrow \neg (t \in S)$
$S \subseteq T$	$\Leftrightarrow (\forall x : S \bullet x \in T)$ Set inclusion.
$S \subset T$	$\Leftrightarrow S \subseteq T \land S \neq T$ Strict set inclusion.
$\varnothing$	The empty set.
$\{t_1, t_2, \ldots, t_n\}$	The set containing the values of terms $t_1, t_2, \ldots, t_n$.
$\{x : T \mid P\}$	The set containing exactly those x of type T for which P holds.
$(t_1, t_2, \ldots, t_n)$	Ordered n-tuple of $t_1, t_2, \ldots, t_n$.
$T_1 \times T_2 \times \ldots \times T_n$	Cartesian product: the set of all n-tuples such that the kth component is of type T_k.
$\mathit{first}(t_1, t_2, \ldots, t_n)$	$== t_1$ Similarly, $\mathit{second}(t_1, t_2, \ldots, t_n) == t_2$, etc.

$\{x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \mid P\}$
The set of all n-tuples $(x_1, x_2, \ldots, x_n)$ with each x_k of type T_k such that P holds.

$\{D \mid P \bullet t\}$ — The set of values of the term t for the variables declared in D ranging over all values for which P holds.

$\{D \bullet t\}$ — $== \{D \mid \mathit{true} \bullet t\}$

$\mathbb{P}\, S$ — Powerset: the set of all subsets of S.

$\mathbb{P}_1\, S$ — $== \mathbb{P}\, S \setminus \{\varnothing\}$
The set of all non-empty subsets of S.

$\mathbb{F}\, S$ — $== \{\, T : \mathbb{P}\, S \mid T \text{ is finite } \}$
Set of finite subsets of S.

$\mathbb{F}_1\, S$ — $== \mathbb{F}\, S \setminus \{\varnothing\}$
Set of finite non-empty subsets of S.

$S \cap T$ — $== \{x : X \mid x \in S \land x \in T\}$
Set intersection.

$S \cup T$ — $== \{x : X \mid x \in S \lor x \in T\}$
Set union.

$S \setminus T$ — $== \{x : X \mid x \in S \land x \notin T\}$
Set difference.

$\bigcap SS$ — $== \{x : X \mid (\forall\, S : SS \bullet x \in S)\}$
Intersection of a set of sets; SS is a set containing as its members subsets of X, i.e. $SS : \mathbb{P}(\mathbb{P}\, X)$.

$\bigcup SS$ — $== \{x : X \mid (\exists\, S : SS \bullet x \in S)\}$
Union of a set of sets; $SS : \mathbb{P}(\mathbb{P}\, X)$.

$\#S$ — Size (number of distinct members) of a finite set.

Numbers

$\mathbb{R}$	The set of real numbers.
$\mathbb{Z}$	The set of integers (positive, zero and negative).
$\mathbb{N}$	$== \{n : \mathbb{Z} \mid n \geq 0\}$ The set of natural numbers (non-negative integers).
$\mathbb{N}_1$	$== \mathbb{N} \setminus \{0\}$ The set of strictly positive natural numbers.
$m \mathrel{..} n$	$== \{k : \mathbb{Z} \mid m \leq k \land k \leq n\}$ The set of integers between m and n inclusive.
$min\ S$	Minimum of a set; for $S : \mathbb{P}_1\, \mathbb{Z}$, $min\ S \in S \land (\forall x : S \bullet x \geq min\ S)$.
$max\ S$	Maximum of a set; for $S : \mathbb{P}_1\, \mathbb{Z}$, $max\ S \in S \land (\forall x : S \bullet x \leq max\ S)$.

Relations

A binary relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations. Let X, Y, and Z be sets; $x : X$; $y : Y$; S be a subset of X; T be a subset of Y; and R a relation between X and Y.

$X \leftrightarrow Y$	$== \mathbb{P}(X \times Y)$ The set of relations between X and Y.
$x\ \underline{R}\ y$	$== (x, y) \in R$ x is related by R to y.
$x \mapsto y$	$== (x, y)$

$\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$

$== \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$

The relation relating x_1 to y_1, x_2 to y_2, ..., and x_n to y_n.

dom R $== \{x : X \mid (\exists\, y : Y \bullet x \underline{R}\, y)\}$

The domain of a relation: the set of x components that are related to some y.

ran R $== \{y : Y \mid (\exists\, x : X \bullet x \underline{R}\, y)\}$

The range of a relation: the set of y components that some x is related to.

$R_1 \mathbin{⨾} R_2$ $== \{x : X;\ z : Z \mid (\exists\, y : Y \bullet x\, R_1\, y \wedge y\, R_2\, z)\}$

Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$.

$R_1 \circ R_2$ $== R_2 \mathbin{⨾} R_1$

Relational composition. This form is primarily used when R_1 and R_2 are functions.

$R^{\sim}$ $== \{y : Y;\ x : X \mid x \underline{R}\, y\}$

Transpose of a relation R.

id S $== \{x : S \bullet x \mapsto x\}$

Identity function on the set S.

R^k The homogeneous relation R composed with itself k times: given $R : X \leftrightarrow X$,

$R^0 = \mathrm{id}\, X$ and $R^{k+1} = R^k \mathbin{⨾} R$.

R^+ $== \bigcup\{n : \mathbb{N}_1 \bullet R^n\}$

$= \bigcap\{Q : X \leftrightarrow X \mid R \subseteq Q \wedge Q \mathbin{⨾} Q \subseteq Q\}$

Transitive closure.

R^* $== \bigcup\{n : \mathbb{N} \bullet R^n\}$

$= \bigcap\{Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \wedge R \subseteq Q \wedge Q \mathbin{⨾} Q \subseteq Q\}$

Reflexive transitive closure.

$R(\!| S |\!)$ $== \{y : Y \mid (\exists\, x : S \bullet x \underline{R}\, y)\}$
Image of the set S through the relation R.

$S \lhd R$ $== \{x : X;\ y : Y \mid x \in S \wedge x \underline{R}\, y\}$
Domain restriction: the relation R with its domain restricted to the set S.

$S \ntriangleleft R$ $== (X \setminus S) \lhd R$
Domain subtraction: the relation R with the elements of S removed from its domain.

$R \rhd T$ $== \{x : X;\ y : Y \mid x \underline{R}\, y \wedge y \in T\}$
Range restriction to T.

$R \ntriangleright T$ $== R \rhd (Y \setminus T)$
Range subtraction of T.

$R_1 \oplus R_2$ $== (\mathrm{dom}\, R_2 \ntriangleleft R_1) \cup R_2$
Overriding; $R_1, R_2 : X \leftrightarrow Y$.

Functions

A function is a relation with the property that each member of its domain is associated with a unique member of its range. As functions are relations, all the operators defined above for relations also apply to functions. Let X and Y be sets, and T be a subset of X (i.e. $T : \mathbb{P}\, X$).

$f\ t$ The function f applied to t.

$X \nrightarrow Y$ $== \{f : X \leftrightarrow Y \mid (\forall\, x : \mathrm{dom}\, f \bullet (\exists_1\, y : Y \bullet x \underline{f}\, y))\}$
The set of partial functions from X to Y.

$X \rightarrow Y$ $== \{f : X \nrightarrow Y \mid \mathrm{dom}\, f = X\}$
The set of total functions from X to Y.

$X \rightarrowtail\mkern-14mu\mapsto Y$ — $== \{f : X \mathrel{+\mkern-10mu\to} Y \mid (\forall y : \operatorname{ran} f \bullet (\exists_1 x : X \bullet x \mathrel{\underline{f}} y))\}$

The set of partial one-to-one functions (partial injections) from X to Y.

$X \rightarrowtail Y$ — $== \{f : X \rightarrowtail\mkern-14mu\mapsto Y \mid \operatorname{dom} f = X\}$

The set of total one-to-one functions (total injections) from X to Y.

$X \mathrel{+\mkern-10mu\twoheadrightarrow} Y$ — $== \{f : X \mathrel{+\mkern-10mu\to} Y \mid \operatorname{ran} f = Y\}$

The set of partial onto functions (partial surjections) from X to Y.

$X \twoheadrightarrow Y$ — $== (X \mathrel{+\mkern-10mu\twoheadrightarrow} Y) \cap (X \to Y)$

The set of total onto functions (total surjections) from X to Y.

$X \rightarrowtail\mkern-18mu\twoheadrightarrow Y$ — $== (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$

The set of total one-to-one onto functions (total bijections) from X to Y.

$X \mathrel{+\mkern-6mu+\mkern-10mu\to} Y$ — $== \{f : X \mathrel{+\mkern-10mu\to} Y \mid f \in \mathbb{F}(X \times Y)\}$

The set of finite partial functions from X to Y.

$X \rightarrowtail\mkern-6mu+\mkern-10mu\mapsto Y$ — $== \{f : X \rightarrowtail Y \mid f \in \mathbb{F}(X \times Y)\}$

The set of finite partial one-to-one functions from X to Y.

$(\lambda x : X \mid P \bullet t)$ — $== \{x : X \mid P \bullet x \mapsto t\}$

Lambda-abstraction: the function that, given an argument x of type X such that P holds, gives a result which is the value of the term t.

$(\lambda x_1 : T_1; \ldots; x_n : T_n \mid P \bullet t)$

$== \{x_1 : T_1; \ldots; x_n : T_n \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$

$\mathrm{disjoint}[I, X]$ — $== \{S : I \mathrel{+\mkern-10mu\to} \mathbb{P} X \mid \forall i, j : \operatorname{dom} S \bullet i \neq j \Rightarrow S(i) \cap S(j) = \varnothing\}$

Pairwise disjoint; where I is a set and S an indexed family of subsets of X (i.e. $S : I \mathrel{+\mkern-10mu\to} \mathbb{P} X$).

$S \ \underline{\text{partitions}}\ T$	$== S \in \text{disjoint} \wedge \bigcup \text{ran}\, S = T$

Sequences

Let X be a set; A and B be sequences with elements taken from X; and $a_1, \ldots, a_n$ terms of type X.

$\text{seq}\, X$	$== \{A : \mathbb{N}_1 \nrightarrow X \mid (\exists\, n : \mathbb{N} \bullet \text{dom}\, A = 1..n)\}$ The set of finite sequences whose elements are drawn from X.
$\text{seq}_\infty\, X$	$== \{A : \mathbb{N}_1 \nrightarrow X \mid A \in \text{seq}\, X \vee \text{dom}\, A = \mathbb{N}_1\}$ The set of finite and infinite sequences whose elements are drawn from X.
$\#A$	The length of a finite sequence A. (This is just '#' on the set representing the sequence.)
$\langle\rangle$	$== \{\}$ The empty sequence.
$\text{seq}_1\, X$	$== \{s : \text{seq}\, X \mid s \neq \langle\rangle\}$ The set of non-empty finite sequences.
$\langle a_1, \ldots, a_n \rangle$	$= \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$
$\langle a_1, \ldots, a_n \rangle \frown \langle b_1, \ldots, b_m \rangle$	$= \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$ Concatenation. $\langle\rangle \frown A = A \frown \langle\rangle = A$.
$\mathit{head}\ A$	The first element of a non-empty sequence: $A \neq \langle\rangle \Rightarrow \mathit{head}\ A = A(1)$.
$\mathit{tail}\ A$	All but the head of a non-empty sequence: $\mathit{tail}\ (\langle x \rangle \frown A) = A$.

$last\ A$	The final element of a non-empty finite sequence: $A \neq \langle\rangle \Rightarrow last\ A = A(\#A)$.
$front\ A$	All but the last of a non-empty finite sequence: $front\ (A \frown \langle x\rangle) = A$.
$rev\ \langle a_1, a_2, \ldots, a_n\rangle$	$= \langle a_n, \ldots, a_2, a_1\rangle$ Reverse of a finite sequence; $rev\ \langle\rangle = \langle\rangle$.
$\frown/\ AA$	$= AA(1) \frown \ldots \frown AA(\#AA)$ Distributed concatenation; where $AA : \mathrm{seq}(\mathrm{seq}(X))$. $\frown/\langle\rangle = \langle\rangle$.
$A \subseteq B$	$\Leftrightarrow \exists\, C : \mathrm{seq}_\infty\ X \bullet A \frown C = B$ A is a prefix of B. (This is just '$\subseteq$' on the sets representing the sequences.)
$squash\, f$	Convert a finite function, $f : \mathbb{N} \ffun X$, into a sequence by squashing its domain. That is, $squash\{\} = \langle\rangle$, and if $f \neq \{\}$ then $squash\, f = \langle f(i)\rangle \frown squash(\{i\} \ndres f)$, where $i = min(\mathrm{dom}\, f)$. For example, $squash\{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\} = \langle A, B, C\rangle$.
$A \upharpoonright T$	$== squash(A \rhd T)$ Restrict the range of the sequence A to the set T.

Axiomatic definitions

Let D be a list of declarations and P a predicate.

The following axiomatic definition introduces the variables in D with the types as declared in D. These variables must satisfy the predicate P. The scope of the variables is the whole specification.

$$\begin{array}{|l} D \\ \hline P \end{array}$$

Generic definitions

Let D be a list of declarations, P a predicate and $X_1, X_2, \ldots X_n$ variables.

The following generic definition is similar to an axiomatic definition, except that the variables introduced are generic over the sets $X_1, X_2, \ldots X_n$.

$$[X_1, X_2, \ldots X_n]$$

$$D$$

$$P$$

The declared variables must be uniquely defined by the predicate P.

Schema Notation

Schema definition

A schema groups together a set of declarations of variables and a predicate relating the variables. If the predicate is omitted it is taken to be true, i.e. the variables are not further restricted. There are two ways of writing schemas: vertically, for example,

$$\begin{array}{|l}
\hline
S \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
x \leq \# y \\
\hline
\end{array}$$

and horizontally, for the same example,

$$S == [x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid x \leq \# y]$$

Schemas can be used in signatures after $\forall$, λ, $\{...\}$, etc.:

$$(\forall S \bullet y \neq \langle\rangle) \Leftrightarrow (\forall x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid x \leq \# y \bullet y \neq \langle\rangle)$$

$\{S\}$	Stands for the set of objects described by schema S. In declarations $w : S$ is usually written as an abbreviation for $w : \{S\}$.

Schema operators

Let S be defined as above and $w : S$.

$w.x$	$== (\lambda S \bullet x)(w)$ Projection functions: the component names of a schema may be used as projection (or selector) functions, e.g. $w.x$ is w's x component and $w.y$ is its y component; of course, the predicate '$w.x \leq \# w.y$' holds.

θS — The (unordered) tuple formed from a schema's variables, e.g. θS contains the named components x and y.

Compatibility — Two schemas are compatible if the declared sets of each variable common to the declaration parts of the two schemas are equal. In addition, any global variables referenced in predicate part of one of the schemas must not have the same name as a variable declared in the other schema; this restriction is to avoid global variables being *captured* by the declarations.

Inclusion — A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must have the same declared sets) and the predicates of S and T are conjoined. For example,

$$\begin{array}{|l} \hline T \\ S \\ z : \mathbb{N} \\ \hline z < x \\ \hline \end{array}$$

is equivalent to

$$\begin{array}{|l} \hline T \\ x, z : \mathbb{N} \\ y : \operatorname{seq} \mathbb{N} \\ \hline x \leq \# y \land z < x \\ \hline \end{array}$$

The included schema (S) may not refer to global variables that have the same name as one of the declared variables of the including schema (T).

Decoration — Decoration with subscript, superscript, prime, etc: systematic renaming of the variables declared in the schema. For example,

S' is
$[x' : \mathbb{N};\; y' : \operatorname{seq} \mathbb{N} \mid x' \leq \# y']$.

$\neg\, S$ The schema S with its predicate part negated. For example, $\neg\, S$ is $[x : \mathbb{N};\; y : \operatorname{seq} \mathbb{N} \mid \neg\, (x \leq \# y)]$.

$S \land T$ The schema formed from schemas S and T by merging their declarations and conjoining (and-ing) their predicates. The two schemas must be compatible (see above).
Given $T == [x : \mathbb{N};\; z : \mathbb{P}\, \mathbb{N} \mid x \in z]$, $S \land T$ is

$$\begin{array}{|l}
\multicolumn{1}{l}{S \land T} \\ \hline
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\, \mathbb{N} \\ \hline
x \leq \# y \land x \in z \\ \hline
\end{array}$$

$S \lor T$ The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. The two schemas must be compatible (see above). For example, $S \lor T$ is

$$\begin{array}{|l}
\multicolumn{1}{l}{S \lor T} \\ \hline
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\, \mathbb{N} \\ \hline
x \leq \# y \lor x \in z \\ \hline
\end{array}$$

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking '$\operatorname{pred} S \Rightarrow \operatorname{pred} T$' as the predicate. The two schemas must be compatible (see above). For example, $S \Rightarrow T$ is

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{S \Rightarrow T} \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\, \mathbb{N} \\
\hline
x \leq \# y \Rightarrow x \in z \\
\hline
\end{array}
$$

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking 'pred $S \Leftrightarrow$ pred T' as the predicate. The two schemas must be compatible (see above). For example, $S \Leftrightarrow T$ is

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{S \Leftrightarrow T} \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\, \mathbb{N} \\
\hline
x \leq \# y \Leftrightarrow x \in z \\
\hline
\end{array}
$$

$S \setminus (v_1, v_2, \ldots, v_n)$

Hiding: the schema S with variables $v_1, v_2, \ldots, v_n$ hidden – the variables listed are removed from the declarations and are existentially quantified in the predicate. The parantheses may be omitted when only one variable is hidden.

$S \upharpoonright (v_1, v_2, \ldots, v_n)$

Projection: The schema S with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden – the variables are removed from the declarations and are existentially qualified in the predicate. For example, $(S \wedge T) \upharpoonright (x, y)$ is

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{(S \land T) \upharpoonright (x, y)} \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
(\exists\, z : \mathbb{P}\, \mathbb{N} \bullet \\
\qquad x \leq \# y \land x \in z) \\
\hline
\end{array}
$$

The list of variables may be replaced by a schema; the variables declared in the schema are used for projection.

$\exists\, D \bullet S$ Existential quantification of a schema.
The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S. The predicate of S is existentially quantified over D. For example, $\exists\, x : \mathbb{N} \bullet S$ is the following schema.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\exists\, x : \mathbb{N} \bullet S} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
\exists\, x : \mathbb{N} \bullet \\
\qquad x \leq \# y \\
\hline
\end{array}
$$

The declarations may include schemas. For example,

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\exists\, S \bullet T} \\
z : \mathbb{N} \\
\hline
\exists\, S \bullet \\
\qquad x \leq \# y \land z < x \\
\hline
\end{array}
$$

$\forall\, D \bullet S$ Universal quantification of a schema.
The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S. The predicate of S is universally quantified over D. For example, $\forall\, x : \mathbb{N} \bullet S$ is the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\forall x : \mathbb{N} \bullet S} \\
\hline
y : \operatorname{seq} \mathbb{N} \\
\hline
\forall x : \mathbb{N} \bullet \\
\qquad x \leq \# y \\
\hline
\end{array}$$

The declarations may include schemas. For example,

$$\begin{array}{|l}
\multicolumn{1}{l}{\forall S \bullet T} \\
\hline
z : \mathbb{N} \\
\hline
\forall S \bullet \\
\qquad x \leq \# y \wedge z < x \\
\hline
\end{array}$$

Operation schemas

The following conventions are used for variable names in those schemas which represent operations, that is, which are written as descriptions of operations on some state,

undashed state before the operation,

dashed state after the operation,

ending in "?" inputs to (arguments for) the operation, and

ending in "!" outputs from (results of) the operation.

The basename of a name is the name with all decorations removed.

ΔS $\quad \widehat{=}\ S \wedge S'$

Change of state schema: this is a default definition for ΔS. In some specifications it is useful to have additional constraints on the change of state schema. In these cases ΔS can be explicitly defined.

ΞS $\quad \widehat{=}\ [\Delta S \mid \theta S' = \theta S]$

No change of state schema.

Operation schema operators

pre S — Precondition: the after-state components (dashed) and the outputs (ending in "!") are hidden, e.g. given,

$$
\begin{array}{|l}
\hline
\quad S \\
x?, s, s', y! : \mathbb{N} \\
\hline
s' = s - x? \land y! = s' \\
\hline
\end{array}
$$

pre S is,

$$
\begin{array}{|l}
\hline
\quad \mathrm{pre}\, S \\
x?, s : \mathbb{N} \\
\hline
\exists\, s', y! : \mathbb{N} \bullet \\
\qquad s' = s - x? \land y! = s' \\
\hline
\end{array}
$$

$S \fatsemi T$ — Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the pairs of after-state components of S and before-state components of T that have the same basename, rename each pair to a new variable, take the conjunction of the resulting schemas, and hide the new variables. For example, $S \fatsemi T$ is,

$$
\begin{array}{|l}
\hline
\quad S \fatsemi T \\
x?, s, s', y! : \mathbb{N} \\
\hline
(\exists\, ss : \mathbb{N} \bullet \\
\qquad ss = s - x? \land y! = ss \\
\qquad \land\, ss \leq x? \land s' = ss + x?) \\
\hline
\end{array}
$$

Index

$\|$, 18
$\mathbb{O}$, 62

Ⓒ, 107
class-union, 62
contain, 109

DC, 118
dcon, 100
dcontain, 105
Δ-list, 13
$\succ\!\!\div$, 65

ec, 130
excon, 131

free, 130
free type, 10
freerefs, 131

inheritance, 14
instantiation, 16

Ⓔ, 132

polymorphic core, 64
polymorphism, 19

scontain, 118
sdcon, 118
sdcontain, 118
secondary attribute and Δ-list, 24
Ⓢ, 119

$\fatsemi$, 19