CORRECTNESS-ORIENTED APPROACHES TO SOFTWARE DEVELOPMENT

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy
in the
Faculty of Science
of
The Queen's University of Belfast

By
Stephen Gilmore, BSc.
September 1990

(Graduation date July 1991)

First printing: April 1991.
Second printing: March 1993.
Third printing: April 1995.

FTP: ftp://ftp.dcs.ed.ac.uk/pub/lfcsreps/91/ECS-LFCS-91-147
WWW: http://www.dcs.ed.ac.uk/lfcsreps/91/ECS-LFCS-91-147

Contents

Dedication

This thesis is dedicated to my mother,
to the memory of my father and to
Bill, Maria and Ben (aged $2\frac{3}{4}$ years).

Acknowledgements

I am very pleased to thank my supervisor Dr. Maurice Clint for his many interesting ideas and for his considerable patience in reading many specifications and proofs in a gaggle of notations and styles.

I would also like to thank four computer scientists who introduced me to, respectively, model-oriented and property-oriented specification styles: David Gray, Peter Kilpatrick, Don Sannella and Andrjez Tarlecki.

The work in this thesis was discussed with several academics who visited the Queen's University of Belfast or who reviewed the author's juvenilia on specifications and refinement. I am grateful for the interest of Ian Hayes, Jim Welsh, Mike Spivey and Joe Morris.

I wish to acknowledge the postgraduate studentship awarded by the Department of Education for Northern Ireland.

Some very wonderful individuals lifted my spirits during this work: Donal Roantree, Marie Owens, Catherine McIlvenna and Elaine Garvin.

Preface

This thesis was submitted for the degree of Doctor of Philosophy at the Queen's University of Belfast in September 1990 and was accepted for that degree in November of that year. I was invited to publish it in this series because I have recently been fortunate enough to obtain a lectureship in the Department of Computer Science at the University of Edinburgh.

I wish to take this opportunity to thank my examiners Bernard Cohen and Peter Kilpatrick for their very detailed study and examination of this thesis.

I also wish to thank my new colleagues at the University of Edinburgh for their welcome. Particular thanks to Jane Hillston and John Spackman.

Abstract

This thesis reports upon the experimental development of a software system. The domain of interest of this study is the use of mathematical reasoning in software development. An experiment is devised in which a modular software system is formally specified in a variety of specification styles. These initial specifications are subsequently refined to efficiently executable implementations. The refinements of the specifications are supported by differing amounts of mathematical reasoning.

The issues to be investigated are the effect of increased use of mathematical analysis in software development and the influence of specification and refinement style on the quality of the subsequent implementation.

Implementation quality is determined by: correctness with respect to the initial formal specification; clarity of the implementation; and machine efficiency. In order of importance, these are ranked: correctness, clarity, efficiency.

The umbrella term used in this thesis to describe software development which employs some degree of mathematical reasoning is 'correctness-oriented development'.

The products of the development are the analysis of specification, refinement and implementation styles and the software system itself.

Part I

Introduction & Concepts

Chapter 1

Introduction

This thesis reports upon the experimental development of a software system. The domain of interest of this study is the use of mathematical reasoning in software development. An experiment is devised in which a modular software system is formally specified in a variety of specification styles. These initial specifications are subsequently refined to efficiently executable implementations. The refinements of the specifications are supported by differing amounts of mathematical reasoning.

The issues to be investigated are the effect of increased use of mathematical analysis in software development and the influence of specification and refinement style on the quality of the subsequent implementation.

Implementation quality is determined by: correctness with respect to the initial formal specification; clarity of the implementation; and machine efficiency. In order of importance, these are ranked: correctness, clarity, efficiency.

The umbrella term used in this thesis to describe software development which employs some degree of mathematical reasoning is 'correctness-oriented development'.

The products of the development are the analysis of specification, refinement and implementation styles and the software system itself.

The software system produced is an efficient software tool which:

- assists authors of formal specifications in communicating their specifications to users of other specification styles; and
- assists in the correctness-oriented refinement of specifications to implementation by calculating properties of the specification.

1.1 Motivation

This thesis is motivated by a desire to judge so-called 'formal methods' in the production of a non-trivial software system under experimental conditions. It is claimed that such methods require more experimental analysis and that more user familiarity with the use of mathematics in the specification, development and maintenance of computer systems is desirable.

Specification languages are used to provide an initial specification of a software system and are also used to document the subsequent production of a satisfactory implementation of the system. In this thesis, the theme of criteria for specification language choice is motivated by the observation that there has of late been a significant increase in the number of specification languages. The proliferation of notations has led to a division of the verification community and a replication of work on similar projects involving the various notations. However, this division and replication should not be considered wasteful. The problems of correct software development are sufficiently pressing and sufficiently difficult to merit study by many computer scientists. Inevitably, these researchers cannot unite under the banner of a single project.

It now seems appropriate to bring together some of the complementary work done in the verification community. The VDM & Z symposium at Kiel in April 1990 indicates that this is the spirit of the age. The forthcoming SafetyNet 90 conference to be held in October 1990 will also bring together results from diverse research camps. This thesis assesses different approaches to the correctness-oriented development of software systems at a time when their correspondences and relationships are of interest to those who are involved in developing formal approaches to software development with the intention of bringing together related approaches.

The analysis of the contrasting specification and refinement styles is achieved by a series of case study developments which record the correctness-oriented development of a modular source-to-source translator for specification languages. The source language of the translator is Z: the target language is the VDM-SL notation of the Vienna Development Method. The modules of the translator are presented as case studies in the chapters of Part 2 of this thesis.

Further evidence of the timeliness of this work was provided at the Z Users Meeting in Oxford on 8th December 1987 (Bowen, 1987) when CAR Hoare proposed an idea for 'a worthwhile research project'. The project was to prove a complete system such as a compiler/loader at a number of levels using such tools and methods as Z (Spivey, 1989), VDM (Jones, 1990), and Unity (Chandy & Misra, 1988). It was suggested that human-readable proofs would be acceptable and that the project would provide an opportunity for exchange of ideas between the development groups involved.

At that time, the research reported in this thesis was already under way. There are several resonances with the more ambitious project proposed by CAR Hoare. Modules of the software are specified using a variety of formal specification notations and are refined to programming language modules using a variety of degrees of rigour in the correctness proofs. No mechanical assistance is employed in the production of the proofs: *only* human-readable proofs are produced.

1.2 Related work

This thesis discusses the relationships between several specification languages and extends this to include the relationship between different approaches to the refinement of specifications to implementations. Although there are many textbooks and journal papers which expound formal specification notations or development methods, there are very few which relate distinct notations or methods.

One textbook which addresses this area is (Cohen *et al.*, 1986) which introduces the reader to algebraic specifications, the Vienna Development Method (VDM) and a Calculus of Communicating Systems (CCS). The same topics are briefly discussed in (Woodcock & Loomes, 1988) although this text deals primarily with the Z notation.

Several technical monographs also provide an introduction to this area. (Sannella, 1988) discusses VDM, Z and algebraic specifications. A comparison of definitions of tree structures in Z and VDM is given in (Achour, 1988b). Proofs of properties of these definitions are given in (Achour, 1988a). An interesting discussion of the use of very high-level languages such as OBJ and Standard ML for specification in conjunction with theorem provers such as REVE and HOL is given in (Stavridou *et al.*, 1987).

A comparison of the VDM specification notation with the programming language OBJ is given in (Duce & Fielding, 1987).

1.3 Thesis structure

This thesis is divided into three parts. The first provides background and introduces and defines terms. The second elaborates the experimental development of a software system. The third presents analysis and conclusions and suggests directions for future work.

1.4 Reader's note

This thesis is divided into a large number of chapters both to ease the task of those required to read and examine it by structuring the presentation of the material and also to make the thesis more useful to casual readers who will be interested in some of the material but not all.

Readers who wish only to obtain an overview of the work should read chapters 5 and 14. Readers familiar with the use of formal methods of program development but who are unsure of the terms which the author may use should also read chapter 2.

Those readers who are interested in comparing different approaches to the specification of software systems should read chapters 6, 9 and 11; those who are interested in constructing proofs of program correctness are referred to chapters 8 and 10; and those interested in using formal specifications to describe formal languages are referred to chapter 12.

1.5 Additional material and implementation

As with any significant endeavour, some of the material generated during the project is not included in this thesis. In particular, the omitted material includes module development reports which provide proofs in more detail than the summaries given in the thesis. Interested readers requiring further details should contact the author.

Readers who wish to obtain a copy of the implementation should contact the author by email using the Internet address `stg@dcs.ed.ac.uk`. Readers who are not connected to the Internet should contact the author by mail to Stephen Gilmore, University of Edinburgh, Department of Computer Science, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK.

Chapter 2

Correctness

This chapter introduces terms which will be used throughout this thesis. It is intended for readers who are already familiar with one school of formal or rigorous software development and wish to acquaint themselves with the terminology which the author will use. Readers who are unfamiliar with formal methods of program development may find this chapter overly terse and are encouraged to read (Jones, 1990) and (Woodcock & Loomes, 1988) before going further.

2.1 Terminology

In "correctness-oriented" software development the delivery of a computer program is accompanied by a proof of its correctness with respect to its formal specification. The correctness proof may take many forms and may be founded on one of many possible logics. It may be 'incomplete': for example, a partial correctness proof does not guarantee termination but ensures that if results are delivered they are correct. Alternatively, the proof may *only* guarantee termination by providing evidence of deadlock freedom.

In another sense, the proof may be incomplete in that some details have been presumed to be obvious and are consequently omitted. Such proofs are termed 'rigorous proofs'. Free-form proofs which contain little or no formal justification are termed 'sketch proofs'.

The possibility of the computer program being delivered with a complete correctness proof is permitted within the scope of correctness-oriented development. Such a proof is termed a 'formal proof'. It consists of an ordered sequence of well-formed formulae each of which is either an axiom of the formal system or follows from preceding well-formed formulae by application of an inference rule of the formal system. Such proofs have the property that they contain sufficient syntactic and semantic detail that they may be machine checked.

In the extreme, a correctness-oriented development may contain no proof text. Such developments contain only two formal texts: the specification of the computer system and the implementation text. The justification for crediting such developments with being correctness-oriented is that a correctness proof could be realised *post facto* after the completion or delivery of the system. The production of an initial formal specification illustrates that the development has taken the correctness of the product as its central concern and development decisions may also have been guided by this principle.

Pathological cases of software development in which programs which claim to be their own specification are produced are not termed correctness-oriented.

Note that implementation text may be available in several forms: source code; semi-compiled object code; and linked executable image. Where the object and image code have been derived mechanically from the source code by compiler and linker programs they are not considered in isolation from the source code. Where the object and image code have been hand-translated as in (Barrett, 1988) they shall be considered to be refinements of the source code which require as much justification as any other development step in correctness-oriented development.

To clarify the concept of correctness-oriented development it is now appropriate to describe the alternative form of software development.

In "production-oriented" software development no formal specification of the delivered computer program is produced. In consequence, there is no attempt to reason mathematically about the correctness of the program. Such developments may be rebuilt as correctness-oriented developments by retrospectively providing a formal specification for the system. This specification may be calculated from the implementation to guarantee correctness of the implementation.

The topics of "production-oriented" software development and the "reverse engineering" process of retrospectively calculating specifications are outside the scope of this thesis. Readers interested in reverse engineering are referred to (Lano & Breuer, 1989).

In the following sections of this chapter the conditions under which software development is termed correct are discussed.

2.2 Correctness conditions

It is not trivial to state the conditions which define the correctness of software systems. In part the difficulty is aggravated by problems at the divide between programs and specifications. In using mathematics to specify software it benefits the specifier to transcend the limitations of the physical device which will ultimately be used to execute a realization of the specification. Specification mathematics offers numbers and sets which may be arbitrarily large: physical computing devices do not. For the present, this difficulty is set aside.

Consider the following specification, S, of a small computer program.

$$S = \{1 \mapsto 2, 2 \mapsto 1, 2 \mapsto 7, 3 \mapsto 6\}$$

This specification is interpreted as the relation between the acceptable program inputs and the corresponding program outputs. The domain of the relation specifies the inputs: the range specifies the outputs. Thus the inverse of S, S^{-1}, describes a different program.

Each input above is related to exactly one output with the exception of 2 which is related to both 1 and 7.

The above specification is correctly implemented by the following program, P.

$$P = \{0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 6\}$$

The program P is expressed as a relation defined by enumeration. This unusual concrete syntax is adequate for the above program although some programs cannot be defined by enumeration and must be described by infinite tables. An implementation which uses a more conventional mix of keywords and operator symbols is given on page 10.

The program P is not contained in its specification, S. The relationship between specification and program is more subtle. The program may extend the domain of the specification. This relationship may be formally stated using set containment.

$$\mathrm{dom}\, S \subseteq \mathrm{dom}\, P \tag{2.1}$$

In the intersection of the domains, the results produced by the implementation P must be acceptable with respect to the specification S. The expression $\mathrm{dom}\, S \lhd P$ below describes the largest subset of P with domain contained in the domain of S.

$$(\mathrm{dom}\, S \lhd P) \subseteq S \tag{2.2}$$

Even allowing for the looseness afforded by non-determinism in the above specification and the freedom for the implementation to extend the domain of the specification, many more implementations are acceptable than the single implementation P presented above. In particular, the *representation* of the inputs and outputs may be altered.

2.3 Representation considerations

An implementation which satisfies 2.1 and 2.2 above will be considered correct even if it differs from the specification in the representation of the inputs and outputs. The required representation of the input may be different from the supplied representation of the output. For simplicity, the case considered here uses the same representation for both. The following representation, A, is used.

$$A = \{0 \mapsto 0_2, 1 \mapsto 1_2, 2 \mapsto 10_2, 3 \mapsto 11_2, 6 \mapsto 110_2, 7 \mapsto 111_2\}$$

The abstraction relation A above is an injective function. No domain element is mapped to two range elements and distinct domain elements are mapped to distinct range elements. A^{-1} is also functional everywhere on its domain and is also injective.

An abstraction relation such as A is termed *adequate*. A representation is provided for all the values used in the specification. The above abstraction relation is *super adequate*: it has a proper sub-relation which is adequate. The following relation A_S is exactly adequate.

$$A_S = (\mathrm{dom}\, S \cup \mathrm{ran}\, S) \lhd A$$

A new specification S_2 may be produced which employs the abstract representation of the inputs and outputs.

$$S_2 = \{1_2 \mapsto 10_2, 10_2 \mapsto 1_2, 10_2 \mapsto 111_2, 11_2 \mapsto 110_2\}$$

This specification is related to the original specification by using the relational composition operator to compose the abstraction function A, the new specification S_2, and the function A^{-1}. The function A^{-1} is known as the *retrieve* function.

$$S = A^{-1} \circ S_2 \circ A$$

An implementation of the specification S_2 may easily be obtained from the implementation of the specification S using the abstraction and retrieve functions.

$$P_2 = A \circ P \circ A^{-1}$$

It is trivial to compute the expansion of this function for the simple program P given earlier. The expansion is given below.

$$P_2 = \{0_2 \mapsto 0_2, 1_2 \mapsto 10_2, 10_2 \mapsto 1_2, 11_2 \mapsto 110_2\}$$

In this context, the abstraction function A is also super adequate. For the implementation an alternative relation strictly contained in A is adequate. Analogous to the case for S, the function A_P below is the smallest adequate abstraction function for P.

$$A_P = (\mathrm{dom}\, P \cup \mathrm{ran}\, P) \lhd A$$

This function is *inadequate* as an abstraction function for the specification S because it fails to provide a representation for the range value 7. Inadequacy is no hindrance to software development if the omitted values are also excised from the possible results returned by the implementation.

The relational composition above may be used as the specification of an implementation I. The composition of relations is used to guide the decomposition of the implementation into separate components which may be implemented in isolation.

2.4 Decomposition

In this section, a more restricted notation will be used to express implementations. The three sub-specifications to be implemented are A_P, P_2 and A_P^{-1}. Only the implementation of P_2 is given here.

The restricted notation used enables the software developer to express an *algorithmic* relationship between the inputs and the outputs of the specification. This is done in order to minimize the number of comparisons required by the naive implementation of P_2 given below.

$$I_2 = \begin{cases} \textsf{if } n = 0_2 \textsf{ then } n := 0_2; \\ \textsf{elsif } n = 1_2 \textsf{ then } n := 10_2; \\ \textsf{elsif } n = 10_2 \textsf{ then } n := 1_2; \\ \textsf{elsif } n = 11_2 \textsf{ then } n := 110_2; \\ \textsf{end if}; \end{cases}$$

A software developer will quickly deduce a simple relationship between the inputs and the outputs and exploit this discovery to reduce the number of comparisons required in the computation of the result. An additional benefit of the deduction is the immediate reduction in the size and complexity of the implementation with a corresponding decrease in object code size.

$$I_2 = \textsf{if } odd(n) \textsf{ then } n := n * 10_2; \textsf{ else } n := n \textsf{ mod } 10_2; \textsf{ end if};$$

There is no reason to consider this to be final description of this computation. The implementation I_2 above may be expressed in a more restricted notation using only tests of least significant bit and left and right shifts.

This completes a brief introduction to correctness. The topics introduced here recur throughout this thesis and more sophisticated examples are to be found in later chapters.

Chapter 3

Specification

This chapter discusses *formal* specifications. Informal specifications are not included in this study. Further, only formal *functional* specifications are discussed. These are exclusively concerned with the behaviour of the system and other considerations such as appearance, efficiency and ease-of-use are not addressed.

A language is said to be formal if it has been defined by a grammar over an alphabet of symbols. For the purposes of machine processing, a formal language must be used to describe computations: machines cannot interpret informal ones. Every programming language is necessarily formal in this sense. Given that formal languages must be used, it remains only to decide when a formal description of the problem should first be given.

The various techniques which are known as *formal methods* all suggest that it is useful to deploy formal languages throughout as much of the process of program development as possible. If software developers are taking a scientific approach to their work, it would seem strange if they were to use anything less.

Clear and exact specification may be obtained by using mathematical abstraction and mathematical data types such as functions and sequences. A library of mathematical definitions is helpful.

It would be highly desirable to be able to insist that customers give software designers a formal statement of their requirements but sadly this is almost always infeasible. However, much of the subsequent communication is between computing professionals, all of whom could become competent in a formal computation description language.

3.1 Specification of datatypes

Consider the specification of a datatype as a component of a larger specification. The datatype taken as an example here is *bitonic sequences*.

A sequence is bitonic if, when considered as a cycle, it has one ascending and one descending sub-sequence. This definition requires clarification before a specification may be given. The following points are noted.

- when considered as a cycle, a bitonic sequence contains no other sub-sequences than the ascending and descending sub-sequences;

- the ascending and descending sub-sequences must be contiguous; and

- either or both sub-sequences may be empty; thus the empty sequence is trivially bitonic.

Taking the familiar definition of a sequence as a partial function from a contiguous prefix of the positive integers to the base type, the first sub-problem to be considered is the cyclic nature of bitonic sequences. The definition of a function which produces the set of all rotations of a sequence is now given.

A rotation of a sequence s is any contiguous sub-sequence of the same length as s which occurs in the concatenation of s with itself—written $s \frown s$. The set of all such rotations is constructed by employing the conjunction of the length restriction and the existence of a prefix and suffix making the rotation equal to $s \frown s$.

$$\begin{aligned}&\mathit{rotations}(s) = \\ &\qquad \{\ b \in \operatorname{seq} X \mid \mathit{len}\ b = \mathit{len}\ s \wedge \exists\, a, c \in \operatorname{seq} X \cdot a \frown b \frown c = s \frown s\ \}\end{aligned}$$

The above definition makes no demands on the elements of type X. Such a definition may be parameterised on X and would then be termed *generic*. This generic definition may be placed in a library of definitions for use in later specifications.

Assuming sets *asc* and *desc* of ascending and descending sequences of elements of type X, a generic definition for the datatype of bitonic sequences of elements of type X may now be constructed.

A bitonic sequence is a sequence which has a rotation which has an ascending prefix and a descending suffix.

$$\begin{aligned}&\mathrm{bitonic}\ X = \\ &\qquad \{\ s \in \operatorname{seq} X \mid \exists\, a \in \mathit{asc} \cdot \exists\, d \in \mathit{desc} \cdot a \frown d \in \mathit{rotations}(s)\ \}\end{aligned}$$

The use of similar constructions to describe operations is addressed in the following section.

3.2 Specification of operations

Consider the specification of an operation to produce a *key word in context* index. Given a set of titles and a set of non-significant words—such as definite articles—the operation is to produce an index relating significant rotations of titles to the titles as given. A significant rotation is any rotation which does not begin with a non-significant word. It can be quickly seen that the index must be a relation by considering two titles which, when rotated, give each other—for example, "International Cinema" and "Cinema International".

The first sub-problem to be considered is the specification of a set of significant rotations of a title. Fortunately, this specification may draw on the specification of bitonic sequences by re-using the specification of the rotation of a sequence.

As stated in the description of a key word in context index, a significant rotation is a rotation which does not begin with a non-significant word. It is simple to state this symbolically.

$$\begin{aligned}&\mathit{sig\_rotations}(\mathit{title}, \mathit{non\_sig}) = \\ &\qquad \{\ s \in \operatorname{seq} \mathit{Word} \mid s \in \mathit{rotations}(\mathit{title}) \wedge \mathit{head}\ s \notin \mathit{non\_sig}\ \}\end{aligned}$$

This said, it only remains to describe a key word in context index as the relation which relates all significant rotations of elements of the title set to the title as given.

$$\begin{aligned}&kwic(s, non\_sig) = \\ &\quad \{\ (rot, title) \in (\mathrm{seq}\ Word)^2 \mid rot \in sig\_rotations(title, non\_sig) \wedge title \in s\ \}\end{aligned}$$

3.3 Specification construction

Constructing specifications is difficult for all but the most elementary problems. It is rarely obvious when embarking on the specification of a problem to decide which are the relevant details and which are the irrelevant ones. Upon completing a specification, several questions should be asked:

- have any relevant details been omitted (is the specification too abstract);
- have any irrelevant details been included (is the specification sufficiently abstract); and
- do the mathematical entities in the specification suitably represent the 'real-world' objects.

Questions are asked throughout the specification process. The impetus to pose pertinent questions and the insight into the problem that the answers give is a large part of the usefulness of writing specifications. Using a formal language forces the specifier to ask more pertinent questions than does the use of natural language as the specification language.

The observations made on page 11 when specifying bitonic sequences are typical products of the questions which are asked during the specification process. The questions seek to establish a set of very simple criteria for describing a bitonic sequence. Flaws of omission are questioned. End cases are found and their acceptability is questioned.

3.4 Specifications and parameters

Parameters make definitions more general and—it is hoped—more useful. A parameterised definition may be used in several different ways throughout the specification and is a more likely candidate for reuse in later specifications than a definition which does not have this flexibility.

Parameterised sub-components are useful in an efficient implementation of a computation but they are significantly more useful in formal, non-executable specifications of computer systems. In the style of specification used in this chapter, datatypes are described in the same notation as operations on typed data. This allows, for example, the specification of the key word in context index to reuse the specification of the rotation of a sequence created for the bitonic sequence datatype. This practice is thought to be useful in allowing greater reuse of specifications. Some authors suggest that an operation/datatype dichotomy serves no purpose in computer science.

The specifications presented in this chapter include little type information. This practice is adopted in this introductory chapter only for simplicity and is not to be encouraged. Set-theoretic specification languages such as Z are founded on typed set theory in order to prevent paradoxes such as those involving self-containing sets. The use of typed formal languages—either specification languages or programming languages—provides much-needed security if the type system is decidable. Typed languages are used throughout the remainder of this thesis. The type systems used are quite sophisticated, providing generic definition mechanisms and identifier and operator overloading, but they remain decidable. Where a definition may be applied to elements of several types the definition will be explicitly labelled as generic.

The use of generic definitions enables typed definitions to be reused in different contexts. In writing specifications, the specifier should be fully aware of the potential for use and reuse of specification components and parameterise the specification where this could allow such reuse.

3.5 Specification languages

The restricted formal languages which software developers use to describe computations and their implementations restrict expression of concepts to a large degree. The use of formal languages at the outset of software development has the potential to shape the remainder of the development through to the final implementation. Restrictions placed on the expression of concepts by specification languages must thus be viewed with more caution than corresponding restrictions on expression in the implementation language of the development.

Thus far in this thesis little attention has been given to the notation which has been used to express the example specifications. Historically, this reflects the evolution of the correctness-oriented development of software. In the seminal works of Hoare (Hoare, 1971; Hoare, 1972), the emphasis is firmly on proof and the notation used to describe the software to be constructed is familiar mathematics. Even in the later work of Bjørner and Jones in (Bjørner & Jones, 1978), the language used for the description of systems is regarded as sufficiently unimportant not to need a name. It is described as "the meta-language", later to be nicknamed Meta-IV and then renamed VDM-SL. The emphasis in these works is on proof and mathematical method. More recently, notations themselves have received more attention.

Chapter 4

Refinement

Refinement is the name given to the activity of developing an efficiently executable implementation from a more abstract specification. Most often, refinements are performed in a number of steps, each of which is a refinement in itself.

A refinement must be *replayable.* That is, it must be recorded in sufficient detail that a path from the initial specification to the final implementation is clearly marked. This documentation of the refinement path enables diagnostic analysis of erroneous refinements and may assist with future developments of related applications.

If significant progress is not made at every refinement step, the record of the refinement becomes lengthy. In an attempt to balance clarity with security, some refinement steps may be defended by sketch or rigorous proofs rather than fully formal proofs. If this is the case, replaying the refinement will require some degree of intelligence. If every step is formally justified, the refinement may be replayed by a machine. With a formal refinement, the impact of changes to the initial specification may be minimized by automated application of the previous refinement to the new specification. This mechanical application may produce a refinement for the new specification. If it does, the new implementation will be correct with respect to the new specification.

Different languages are often used at different stages of software development: specification, design and implementation. Links between related objects from different stages may be recorded using a *refinement language* such as the notation of natural deduction proofs (Jones, 1990).

Refinement is a topic of current research. In particular, the refinement of concurrent systems engenders many difficult problems. However, those problems fall outside the scope of this thesis. Here, the principal concern is development of sequential software.

4.1 Algorithmic refinement

An interesting topic is the representation of algorithms in specifications and subsequent refinements. In an initial, very abstract specification, virtually no algorithmic detail is included. In the final, executable refinement of this specification none is omitted. The question arises as to how and when is algorithm introduced?

Consider the familiar problem of sorting a sequence of natural numbers into ascending order. A Z specification of the *sort* function is given below. The essential, simple properties of sorting are stated formally: the result is a permutation of the argument and the result is an ascending sequence.

$$
\begin{array}{l}
sort : \operatorname{seq} \mathbb{N} \rightarrow \operatorname{seq} \mathbb{N} \\
\hline
\forall s : \operatorname{seq} \mathbb{N} \cdot \\
\quad items\ (sort\ s) = items\ s \wedge \\
\quad (\forall i, j : \operatorname{dom}\ (sort\ s) \cdot i < j \Rightarrow sort\ s\ i \leqslant sort\ s\ j)
\end{array}
$$

The *items* function converts a sequence into a bag—also called a multiset—which contains all the elements of the sequence with the same frequency of occurrence but without any record of the order in which they occurred in the sequence.

The above specification elegantly and exactly captures the required properties of sorting. As a specification, it embodies the essential qualities of abstraction, clarity and brevity that initial specifications should exhibit. Evidently, no practical algorithm is suggested by the specification.

A direct implementation of the above specification would randomly generate an untyped result which could be tested to discover if it:

1. represents a sequence of natural numbers;

2. permutes the argument of the function; and

3. is an ascending sequence.

If the randomly generated result passes all three tests, then the—in this case, unique—result has been found. If not, another result is randomly generated and the process is repeated. Such a *nondeterministic algorithm* would not be accepted as a satisfactory implementation of the *sort* function. The specification must evidently be refined.

Given the specification of *sort* above, a software developer would be at liberty to choose a familiar sorting algorithm considering criteria such as knowledge of some existing ordering in the sequence. Alternatively, the developer may choose to produce an original sort which exploits a known existing order in the sequence.

Consider the more usual case where no known ordering is present and take Quicksort as the sorting algorithm. In Quicksort, input sequences of more than one element are partitioned into two non-empty sequences. Both sequences are then sorted by Quicksort. These sorted sequences are then concatenated in the correct order to produce a single sorted sequence which is a permutation of the initial input.

Begin by defining a *partition* function which is a partial function from a non-empty sequence of natural numbers to a sequence of exactly two non-empty sequences of natural numbers. The elements of the argument are distributed among the two sequences. No element in the first sequence may exceed any element in the second. The function is partial because the sequence argument must contain at least two elements; this cannot be expressed using the standard Z datatypes and a definition of such a datatype would make the specification unwieldy.

$$
\begin{array}{l}
partition : \operatorname{seq}_1 \mathbb{N} \rightarrow\!\!\!\!\!\!+\ \operatorname{seq}_1(\operatorname{seq}_1 \mathbb{N}) \\
\hline
\forall s : \operatorname{seq}_1 \mathbb{N} \cdot \\
\quad \#(partition\ s) = 2 \wedge \\
\quad items\ ({}^\frown/(partition\ s)) = items\ s \wedge \\
\quad max\ (\operatorname{ran}\ (partition\ s\ 1)) \leqslant min\ (\operatorname{ran}\ (partition\ s\ 2))
\end{array}
$$

The distributed concatenation operator denoted by '$\frown/$' produces a sequence from a sequence of sequences by concatenating the sequence elements in the order in which they appear in the sequence of sequences.

The *quicksort* function is defined to be a total function over sequences of natural numbers. For any sequence of one element or less, the function returns its argument. Sequences of two elements or more are partitioned using *partition*. The partitions are then sorted using recursive applications of *quicksort* and the resulting sequences are concatenated.

$$
\begin{array}{|l}
quicksort\colon \operatorname{seq} \mathbb{N} \rightarrow \operatorname{seq} \mathbb{N} \\
\hline
\forall\, s\colon \operatorname{seq} \mathbb{N} \cdot \\
\qquad \#s \leqslant 1 \Rightarrow quicksort\ s = s \wedge \\
\qquad \#s > 1 \Rightarrow quicksort\ s = \frown/(partition\ s \mathbin{;} quicksort)
\end{array}
$$

The relational operator denoted by '$\mathbin{;}$' is forward functional composition. It produces a sequence of the same length as its sequence operand. The elements of the result are the images of corresponding elements of the given sequence under the function operand.

The *quicksort* function is a correct refinement of the *sort* function. (The interested reader is referred to (Gilmore & Clint, 1987) for proof of this result and also for similar refinement treatments of Heapsort, Insertion Sort, Bubblesort and Selection Sort.) The specification of *quicksort* resembles a recursive function definition in a functional programming language—albeit in an unconventional concrete syntax. A base case and a self-referential case are provided and a subordinate function is used to structure the presentation of the function. The forward functional composition operator is provided in many functional programming languages as a standard polymorphic list processing operation.

The subordinate *partition* operation is not conventionally executable and requires further refinement. Partitioning may be accomplished by selecting a pivot and separating the remaining elements into two sequences by successively comparing each with the pivot. The two subsequences are constructed as 'accumulation parameters' of a further subordinate function called $partition_2$.

$$
\begin{array}{|l}
partition\colon \operatorname{seq}_1 \mathbb{N} \nrightarrow \operatorname{seq}_1(\operatorname{seq}_1 \mathbb{N}) \\
\hline
\forall\, s\colon \operatorname{seq}_1 \mathbb{N} \cdot partition\ s = partition_2(head\ s, tail\ s, \langle\rangle, \langle\rangle)
\end{array}
$$

The $partition_2$ function compares a pivot, p, with the elements of a sequence s. The elements of s which are not larger than p are added to the accumulation sequence s_1. Elements of s larger than p are added to s_2. It does not matter where the appropriate elements are placed in the accumulation sequences; the important quality is that s_1 and s_2 partition s. Choose to append the elements in both cases. Finally, the pivot must be placed in one of the subsequences in order to ensure that both are non-empty.

$$
\begin{array}{|l}
\mathit{partition}_2 \colon \mathbb{N} \times \mathrm{seq}\,\mathbb{N} \times \mathrm{seq}\,\mathbb{N} \times \mathrm{seq}\,\mathbb{N} \nrightarrow \mathrm{seq}_1(\mathrm{seq}_1\,\mathbb{N}) \\
\hline
\forall\, p \colon \mathbb{N};\ s, s_1, s_2 \colon \mathrm{seq}_1\,\mathbb{N} \cdot \\
\quad (s = \langle\rangle \Rightarrow \\
\quad\quad \mathit{partition}_2(p, s, s_1, s_2) = \mathit{place}(p, s_1, s_2)) \land \\
\quad (s \neq \langle\rangle \land \mathit{head}\ s \leqslant p \Rightarrow \\
\quad\quad \mathit{partition}_2(p, s, s_1, s_2) = \mathit{partition}_2(p, \mathit{tail}\ s, s_1 \frown \langle \mathit{head}\ s\rangle, s_2)) \land \\
\quad (s \neq \langle\rangle \land \mathit{head}\ s > p \Rightarrow \\
\quad\quad \mathit{partition}_2(p, s, s_1, s_2) = \mathit{partition}_2(p, \mathit{tail}\ s, s_1, s_2 \frown \langle \mathit{head}\ s\rangle))
\end{array}
$$

The *place* function constructs the sequence of sequences from the pivot and the two accumulation sequences. At most one of the accumulation sequences should be empty. The pivot replaces an empty sequence if one exists, otherwise it is added to the second sequence argument. As before, the exact placement is unimportant. Below, the pivot is appended.

$$
\begin{array}{|l}
\mathit{place} \colon \mathbb{N} \times \mathrm{seq}\,\mathbb{N} \times \mathrm{seq}\,\mathbb{N} \nrightarrow \mathrm{seq}_1(\mathrm{seq}_1\,\mathbb{N}) \\
\hline
\forall\, p \colon \mathbb{N};\ s_1, s_2 \colon \mathrm{seq}_1\,\mathbb{N} \cdot \\
\quad (s_1 = \langle\rangle \Rightarrow \mathit{place}(p, s_1, s_2) = \langle\langle p\rangle, s_2\rangle) \land \\
\quad (s_1 \neq \langle\rangle \Rightarrow \mathit{place}(p, s_1, s_2) = \langle s_1, s_2 \frown \langle p\rangle\rangle)
\end{array}
$$

This refinement of the *partition* function has produced a hierarchy of functions all of which eschew sophisticated use of predicates for specification in favour of recursive or conditional definitions. Such use of the Z notation would be considered too algorithmic for an initial, abstract specification although it is acceptable in a later refinement.

The use of the Z notation for refinement may be unfamiliar and somewhat unpleasing to some readers. It is noted that Z is most often used for initial, very abstract specifications. The language intentionally does not provide constructs which are familiar from conventional programming languages. Mathematical symbols are used in place of keywords to heighten the intended difference in use.

The absence of constructs such as conditional expressions makes direct, algorithmic function definitions overlong. Compare the above Z definition of $\mathit{partition}_2$ with the following VDM-SL definition where the repeated use of implication is replaced by nested conditional expressions.

$$\mathit{partition}_2 : \mathbb{N} \times \mathbb{N}^* \times \mathbb{N}^* \times \mathbb{N}^* \to \mathbb{N}^{**}$$

$$
\begin{array}{lll}
\mathit{partition}_2(p, s, s_1, s_2) & \triangle & \mathsf{if}\ s = [\,] \\
 & & \mathsf{then}\ \mathit{place}(p, s_1, s_2) \\
 & & \mathsf{else\ if\ hd}\ s \leqslant p \\
 & & \quad \mathsf{then}\ \mathit{partition}_2(p, \mathsf{tl}\, s, s_1 \curvearrowright [\mathsf{hd}\, s], s_2) \\
 & & \quad \mathsf{else}\ \mathit{partition}_2(p, \mathsf{tl}\, s, s_1, s_2 \curvearrowright [\mathsf{hd}\, s])
\end{array}
$$

The repetition of conditions and their negations is avoided by the use of the conditional expression. Although keywords are used in VDM-SL, their use is not essential for expressing conditional definitions. An earlier version of Z used in (Sufrin, 1982) allowed the construct $p \longrightarrow p_1, p_2$ as a shorthand for $(p \land p_1) \lor (\neg p \land p_2)$.

The discovery that VDM-SL may be more elegant than Z for the expression of algorithmic designs is due in part to the broad outlook adopted by its designers. In (Jones, 1980, page 116), C.B. Jones dismisses programming languages for the documentation of algorithms and advocates use of specification languages for this purpose.

> It should be clear from what has been said so far that a programming language is not a viable tool for algorithm documentation. The use of such a language forces commitments which are irrelevant to the essence of an algorithm.

Now the question at the beginning of this section may be at least partly answered. Algorithm is introduced into specifications through structuring and through simpler, restrained use of language features which will allow introduction of design language constructs such as the conditional expression of VDM-SL and, ultimately, the introduction of implementation language constructs such as the conditional statement of imperative programming languages.

4.2 Refinement and complexity

The above example refinement illustrates well the principles of refinement. A sequence of successively more detailed descriptions of sorting is produced and marked progress towards an efficiently executable implementation is made at every refinement step. Refinement steps can be connected by proof text. However, it is unconvincing as a genuine example of refinement since the author could picture an implementation of Quicksort at the outset of the refinement. Thus, no *discovery* was needed in the refinement.

Not all refinements will succeed. It is possible to specify non-computable problems such as the halting problem. It is also possible to produce a specification which cannot be refined to implementation because of contradictions in the specification. Even if the specification has a computable solution, no polynomial-time algorithm may be achievable. This would limit the usefulness of an implementation since it could only be applied to relatively small instances of the problem. Alternatively, a polynomial-time solution may be possible but might be as yet unknown. Two further examples are used to illustrate refinement: a "pattern matcher" is specified in VDM-SL and the "stable room-mates"problem is specified in Z.

4.2.1 A pattern matcher

A pattern matcher is a program which decides if a sequence of characters conforms to a given pattern. Patterns contain strings and *jokers*—the wild cards of the pattern. The following is a pattern with jokers X, Y and Z.

$[\text{“It is a”}, X, \text{“to”}, Y, \text{“—”}, Z]$

This pattern is matched by the following strings.

♡ "It is a luxury to be understood. — EMERSON."

♡ "It is a great art to saunter. — THOREAU."

♡ "It is a mistake to speak of a bad choice in love, since, as soon as the choice exists, it can only be bad. — PROUST."

The pattern is not matched by the following string because it does not contain the "to" string.

♠ "It is a very sad thing that nowadays there is so little useless information.
— WILDE."

Patterns may involve several occurrences of a joker—as in $[X, \text{“+”}, Y, \text{“=”}, X, \text{“*”}, Z]$. A matching string must contain two occurrences of the same substring at the places where X occurs in the pattern.

Define a pattern to be a sequence of elements which are either strings or jokers.

$$Pattern = Element^*$$

$$Element = String \cup Joker$$

$$String = CHAR^*$$

Observing that strings in the pattern must appear unchanged in any matching string and that each occurrence of a joker must map to the same string, it may then be noted that patterns and matching strings may be associated via functions from elements to strings which are identities for strings. Call these *Expand* functions.

$$Expand = Element \xrightarrow{m} String$$

where

$$inv\text{-}Expand(e) \quad \triangleq \quad \forall s \in String \cdot e(s) = s$$

Define an *apply* function which applies an expand function to each element of a sequence to return the forward functional composition of the sequence and the function.

$$apply : Pattern \times Expand \rightarrow String^*$$

$$apply(p, e) \quad \triangleq \quad \mathsf{if}\ p = [\,]\ \mathsf{then}\ [\,]\ \mathsf{else}\ [e(\mathsf{hd}\, p)] \frown apply(\mathsf{tl}\, p, e)$$

A pattern p is matched by a string s if there is an expand function which can be used to convert p to s. These are then concatenated in order using the VDM-SL distributed concatenation operator, 'dconc'.

$$matches : String \times Pattern \rightarrow \mathbb{B}$$

$$matches(s, p) \quad \triangleq \quad \exists e \in Expand \cdot \mathsf{dconc}\, apply(p, e) = s$$

This specification is extremely compact. The subordinate *apply* function has been specified as a direct definition and is obviously implementable. However, the *matches* function proves more difficult to implement. This example better illustrates the difficulties of refinement than the sorting example since—for many computer scientists—pattern matching algorithms are less well known than sorting algorithms. For this author at least, there is no obvious first refinement step to be taken when attempting to refine the above specification to an efficient implementation.

4.2.2 The stable room-mates problem

In a certain university, overcrowding is so severe that the lecturers are forced to share offices—two lecturers to each office. Each year the offices are re-allocated. Some lecturers do not wish to share with certain others so the office administrator asks them to supply a list of preferences for a room-mate. All the possibilities must be ranked from favourite down to least favourite. All the lecturers would rather have an office of their own and always rank themselves first. The office administrator tries to find a set of pairings of lecturers which is stable. The allocation is said to be stable if no two lecturers would prefer each other to their current room-mates.

Datatypes

From the statement of the problem, an even number of lecturers are involved. Let this number be n. Let the lecturers be numbered from 1 to n. The lists of preferences are total finite bijective functions from lecturers to a rank from 1 to n. A table of the preferences is constructed—one list for each lecturer.

$$Person == 1 \mathrel{..} n$$

$$Table == Person \rightarrow (Person \rightarrowtail\!\!\!\!\twoheadrightarrow 1 \mathrel{..} n)$$

Operations

The initial table is constructed with each lecturer, p, ranking p as first preference.

$$\begin{array}{|l}
\text{\underline{ Initial_Table }} \\
prefs : Table \\
\hline
\forall p : Person \cdot prefs\ p\ p = 1 \\
\hline
\end{array}$$

Additional datatypes

There are two possible results for the problem, either a stable configuration can be found or it cannot. If a stable configuration cannot be found, no further information need be supplied. If a stable configuration can be found, an allocation list is produced. Together, the allocation list and the result are the state of the system.

$$Allocation == Person \rightarrowtail\!\!\!\!\twoheadrightarrow Person$$

$$Result ::= unstable \mid stable \langle\!\langle Allocation \rangle\!\rangle$$

In addition to being a bijective function, the allocation list a must be symmetric. That is, if p_1 shares with p_2 then p_2 must share with p_1. Of course, p_1 and p_2 must be different people otherwise there will not be enough offices to go round. That is, the allocation list must be anti-reflexive.

$$\begin{array}{|l}
\text{\underline{ Rooms }} \\
a : Allocation; \\
r : Result \\
\hline
\forall p_1, p_2 : Person \mid p_1 \neq p_2 \cdot (a\ p_1 \neq p_1) \wedge (a\ p_1 = p_2 \Leftrightarrow a\ p_2 = p_1) \\
\hline
\end{array}$$

Additional operations

Now the criteria for stability may be specified. For any pair of lecturers p_1 and p_2, p_2 must be ranked lower in p_1's list of preferences than p_1's current office-mate or contrariwise for p_2.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Stable}} \\
\Xi \textit{Initial\_Table} \\
\Delta \textit{Rooms} \\
\hline
\forall\, p_1, p_2 : \textit{Person} \cdot \\
\quad (\textit{prefs}\ p_1\ p_2 \geqslant \textit{prefs}\ p_1\ (a'\ p_1)) \lor (\textit{prefs}\ p_2\ p_1 \geqslant \textit{prefs}\ p_2\ (a'\ p_2)) \\
\hline
\end{array}
$$

An unstable system is one where no such a' exists.

$$\textit{Unstable} \mathrel{\widehat{=}} \forall\, a' : \textit{Allocation} \cdot \neg\ \textit{Stable}$$

A success report is delivered if a stable configuration has been found. A failure report is delivered otherwise.

$$\textit{Success} \mathrel{\widehat{=}} [\Delta \textit{Rooms} \mid r' = \textit{stable}(a')]$$

$$\textit{Failure} \mathrel{\widehat{=}} [\Delta \textit{Rooms} \mid r' = \textit{unstable}]$$

$$\textit{Room\_Mates} \mathrel{\widehat{=}} (\textit{Stable} \land \textit{Success}) \lor (\textit{Unstable} \land \textit{Failure})$$

This example illustrates some further issues in specification and refinement and another reason for producing refinements. When first presented with this problem, a wary software developer will first attempt to determine that the implicit decision problem is decidable, and then to show that the entire problem has a computable solution. Both questions give favourable answers: the decision problem is decidable and the entire problem has a computable solution. The next information that the cautious software developer would require is the complexity class of the problem. If the problem is intractable, it may not be worthwhile to proceed with the development.

The tractability or intractability of the problem cannot be determined from the specification. This is another reason why computer scientists perform refinements of specifications: to discover the complexity class of interesting problems.

It is not obvious whether or not the stable room-mates problem is solvable in polynomial time. A polynomial time algorithm has recently been found.

This concludes a brief introduction to refinement. As noted at the start of this chapter, more will be said about refinement in the future. The theme recurs throughout this thesis.

Part II

A Correctness-Oriented Development

Chapter 5

A Specification Translator

This chapter provides an overview of the correctness-oriented development with which this thesis is mainly concerned. The first section explains the reasoning which led to the choice of subject area for the development. Following this, the structure, specification and implementation of the specification source-to-source translator are discussed. Some details of the translation are then provided. Possible alternative designs for the translator are then discussed. The chapter concludes with further details of the implementation.

5.1 Choice of implementation problem

Several criteria were defined for the selection of the implementation problem for this experimental development.

1. The finished system should be a useful and efficient software tool.

2. The implementation problem must be of sufficient size to require correctness-oriented development but should not unreasonably burden a single software developer.

3. The software tool produced should assist or promote use of mathematical reasoning in software development.

4. No significant re-implementation or replication of existing software should take place in the development.

5. The product should be modular and constructed from general, re-usable components.

The above criteria influenced many factors of the subsequent development. The number of possible subjects was reduced by 1–4. The choice of implementation language was guided by 1 and 5. The quantity of supporting proof text that was produced was limited by 2.

All but one of the above criteria are pragmatic software engineering principles and need no further defence here. The decision to require that the software tool produced be such that it assisted or promoted the use of mathematical reasoning in software development requires further explanation.

It is thought by some that cost-effective use of formal methods requires machine assistance. Whether this is or is not the case, support tools for formal approaches have been produced but are not readily available to many users. Some remain the objects of research investigation and others are proprietary tools which are not for distribution or commercial sale. A counter-example is the *f*UZZ type-checker for Z (Spivey, 1988b) which is commercially available and supported by the author. The area of software support for formal methods was identified as a subject area which merited further attention. Other colleagues have also addressed this area (McParland, 1989).

It is widely held that support tools for formal methods should be developed using formal approaches. Machine assistance may of course be employed in the development of a support tool. In the case of J.-R. Abrial's B Tool (Abrial, 1988), an earlier version of the B Tool is being used in the formal re-implementation of the B Tool itself. The desire to encourage the use of formal approaches when developing support tools for formal methods was a factor in choosing for this experiment the problem of implementing of a software tool to support formal methods.

It was decided that the support tool should not favour one formal method over another since these methods would be used in the experimental development of the tool and such bias would not reflect well on the fairness of the experiment. A translation system would enable two such approaches to be promoted in addition to providing a classical experimental development subject. The translation system would be a source-to-source translator for specifications. The source and target languages were chosen to be Z and VDM-SL respectively.

5.2 Structure of the translator

The translator is composed of many small modules but comprises three large components.

- A Z parser and tree-builder;
- a tree-processing translator;
- a VDM-SL unparser.

In order to minimize the programming task, several compromises were struck regarding the behaviour of the translator. The first was the decision that only type-checked Z specification texts would be accepted as input. Given a syntactically or semantically incorrect Z text, the translator could behave in any way: for example, it might halt with a run-time error message or produce any VDM-SL text. There is no obligation on the translator to report syntax or type errors in the input. This decision to omit error detection and recovery considerably simplifies the coding of the Z parser.

At the outset of the project no type-checker for Z existed but it was considered likely that one would be developed before the completion of the project. Spivey's *f*UZZ package became commercially available in 1989.

The decision not to enforce the syntax and type rules also provided some flexibility to modify the grammar by which the input is parsed. The input grammar was weakened and generalized in order to ease the task of parsing. The transformations made to the grammar ensured that any sentence generated by the old grammar can also be generated by the new grammar although the converse is not true.

A second compromise was the decision to fix on a version of the Z basic library. The version chosen is, with some omissions, that given in (Spivey, 1989).

The idea of having a modifiable library of definitions is appealing. It allows Z to be readily used in different application areas by defining datatypes and standard functions which then become the primitive operations of the specifications in these areas.

Unfortunately, the use of a basic library makes machine processing of Z specifications more difficult. Elegant translation from Z to VDM-SL requires using properties of the datatypes and relations which are defined in the Z basic library. These properties are non-trivial: they cannot be deduced by the translator. In order to produce more readable VDM-SL output, the translator was designed for a fixed version of the library.

A third compromise was the decision to choose a relatively simple dialect of Z as the input language of the translator. The dialect chosen is explained below.

5.2.1 The input language

There are many excellent textbooks and technical monographs which describe Z from various viewpoints, for example: (Hayes, 1987; Spivey, 1988a; King *et al.*, 1988; Woodcock & Loomes, 1988; Spivey, 1989). However, none of these claims to be an authoritative standard for the Z language. This is commendable: Z is free to change and to be improved.

This freedom has been exploited to simplify the translator by choosing a dialect of Z whose semantics is closer to that of the subset of Z given in (Spivey, 1988a) than most other dialects. It is also easier to parse than some other dialects. The more recent of the works referred to above had significant influences on the input language.

The dialect of Z defined in (King *et al.*, 1988) was used to define the concrete syntax of the input language but the grammar used was based on that given in (Spivey, 1989).

The input language also omits some concepts which are offered in some dialects of Z but not in others. These include the data type 'iseq' of injective sequences, free type definitions and the mechanism for importing definitions from Z documents in a specification library. These decisions reduced the size of the language while retaining the most familiar features. Even after this pruning, the input language is not trivial.

5.2.2 The output language

The dialect of VDM-SL generated is from the school of VDM known as 'the British style' and typified by C.B. Jones' books (Jones, 1980; Jones, 1990). At present, there is no definitive standard for VDM-SL although work on a British standard is under way and an agreed syntax is included as an appendix of (Jones, 1990).

The concrete syntax of the output language generated by the translator is the dialect from (Wolczko, 1990) which is acceptable to the LaTeX typesetting system. Thus, the output from the translator is typeset VDM-SL text which is ready for printing on a variety of output devices. A large character set is made available in this way and the mathematical symbols of VDM-SL such as $\neg$, $\wedge$, $\vee$, $\in$, $\forall$ and $\exists$ are printed as here.

As might be expected, the VDM-SL output generated by the translator will typically not use the features of VDM-SL as elegantly as a specification which had been written in VDM-SL initially. Although every attempt is made to generate readable VDM-SL output, a skilled VDM-SL author could frequently improve the clarity of the specification.

Another reason for choosing the dialect of VDM-SL from (Wolczko, 1990) was that it is accepted by the STC VDM-SL structure editor and subsequent enhancement of the specification can therefore take place in an environment which is designed to aid this task.

5.3 The specification of the translator

It is common in the literature on formal specification to specify a parser by giving a formal description of the language to be parsed. However, it is clearly not sufficient to describe a translator by giving formal descriptions of the source and target languages. The mapping of input constructs onto output constructs must also be defined.

A natural modular approach to the specification of a translator is to map sentences of the source and target languages into a neutral domain such as the world of typed sets. Translation may then be specified by equating the images of the input and output under the appropriate mappings.

An alternative approach was adopted for this specification. The above approach would suggest two large modules: one for the definition of the mapping of the source language into the neutral domain; the other for the definition of the mapping of the target language into the neutral domain. Greater modularity was required in the specification because a large number of combinations of ⟨specification notation, degree of rigour⟩ pairs were to be compared in the development. The implementation of the translator was also required to proceed as modules of the specification were written. The increased modularity meant that the development of the implementation would not lag too far behind the production of the specification.

The first specifications written for modules of the system were expressed in Z. Later specifications were written in VDM-SL. The final specifications were written using an algebraic notation. The pairings of modules and specification notations is now given.

- Z—Source handler; lexical analyzer; translator.
- VDM-SL—Symbol table; parser.
- Algebraic—Table builder.

5.4 Approaches to implementation

Several strategies were used to create implementations satisfying the specifications. The wide range of techniques used means that the development of the translator cannot be termed 'formal' or even 'rigorous'. At best, it could be said to be 'correctness-oriented'.

Caution: It is not suggested that 'correctness-oriented' development is a desirable approach to software development. It is only appropriate in this case because of the experimental nature of the development and because the consequences of the failure of the resulting software product are not grave. The reader is cautioned that the products of this style of development should not be termed 'verified' or 'correct': this style of software development does not demand as much proof as a completely rigorous or a wholly formal development and cannot be expected to produce the same results.

The modules of the program are being tested by proof and never by execution. In some cases, no proof text has been constructed to link a module of the specification and a proposed implementation. Nothing can be said about the fitness of those modules to perform the required computation.

Of greater interest are the cases where proof text in various styles has been constructed. These are now discussed.

5.4.1 A natural language style

The first proof style used was natural language based. In this style, invariants of loops and datatypes were stated formally in the Z notation but proofs of invariance were merely sketched by giving English descriptions of proof details. It was hoped that this style would be as convincing as a more symbolic proof while avoiding the 'formal clutter' of some proof styles. It was also hoped that the assumption of reasonable intelligence on the part of the reader would allow much shorter proofs to be written.

5.4.2 A rule-based style

A more economical style of program development was sought. In contrast to the previous style which was *post facto* verification, this style would be calculation and transformation (Broy, 1983) based.

Sets of specialized rules were created for the refinement of the specification. These rules were tailored to the author's specification style to allow them to be kept as simple as possible and to allow the most immediately useful set of rules to be created. The rules fall into three obvious categories:

- specification transformation rules;
- specification to program transformation rules;
- program transformation rules.

Of these, the most powerful rules are the specification transformation rules. These rules assist the creation of a 'program-like' Z text. The specification to program transformation rules ease secure translation into a programming language notation. The program transformation rules provide simple mechanisms to re-shape the implementation in order to clarify program structure rather than to improve the efficiency of the program.

5.4.3 A systematic style

The next development style was systematic (Jones, 1990) in character. Here, details of data reifications were recorded as abstraction functions but the attendant proof obligations of adequacy and the proof obligations concerning the operational behaviour of the specification were not discharged.

5.4.4 A rigorous style

The next development style employed was classical VDM rigorous development. The development proceeded by data reification and operation decomposition with the attendant proof obligations of satisfaction, adequacy and even implementability being discharged in the natural deduction **from** .. **infer** style of (Jones, 1990).

In some ways, this style of development was the least taxing. Indecision about which properties of the system to prove was removed, making development less hesitant.

The decision to discharge all the relevant proof obligations meant that the author was left with an extremely long proof text which was then summarized to increase its readability. In the author's opinion, the strategy of summarizing a rigorous proof produced one of the most elegant proof texts of the development.

5.5 Development policy

Several specification and refinement styles are used throughout the development of the specification translator in order to give a controlled comparison of their relative merits. It was the author's policy that no attempt should be made to choose an appropriate specification notation for each module of the translator. Instead, a notation is chosen and several subsequent modules are specified in this style; the notation is changed and several more modules are specified.

There are two reasons for this choice. First, it is assumed that each of the notations should be adequate to specify any of the modules of the translator and, more generally, they should be adequate to specify any software system. If this assumption is incorrect, the development may uncover such limitations which may, of themselves, be interesting to report. The second reason for the above choice is the desire to limit the amount of detailed specification language knowledge to be retained throughout the development by phasing the need for expertise in each notation. It is also hoped that factoring the development into a Z phase, a VDM-SL phase and an algebraic phase has made this thesis easier to read than if the alternative had been chosen.

Only one specification notation is used for each module and each merits a separate chapter. Other than the start of a new chapter, no announcement of a change of notation is given. However, the three notations used are visually distinct. Z is described as 'boxed mathematics' and specifications are enclosed in boxes or highlighted by visual formatting effects. VDM-SL uses a mix of keywords and mathematical symbols and specifications are displayed as paragraphs of mathematical text separated by natural language commentary. Algebraic specifications consist of modular algebras containing long sequences of conditional equations.

5.6 Details of the translation

The details of the translation strategy address in the main the principal differences between the languages Z and VDM-SL. These differences are the interpretation and selection of predicates to constrain initial and final values of variables and the structuring facilities available for combining operations. Definitions of typed variables and predicates are common to both languages.

The majority of the work in translation is in re-shaping the operations and user-defined datatypes. The standard functions and pre-defined datatypes may be appropriated from the source language or defined generically in the target language.

5.6.1 Schemas

Schemas are used for two purposes in Z. They are used to define datatypes and also to define operations. For example, the schema *Coord* below defines a datatype of co-ordinates and the *Home* schema sets the co-ordinate c to the origin $(0, 0)$ using Z's unique value μ function.

Coord

$x, y : \mathbb{N}$

Home

$c, c' : Coord$

$c' = (\mu\ Coord \mid x = 0 \wedge y = 0)$

The distinction between the purposes of the two schemas is made by inspecting the form of their signatures. Schemas which contain only undecorated variables in their signature are translated as datatypes; schemas which contain decorated variables are translated as operations.

The following VDM-SL translation would be produced. The two variables c and c' are resolved into one external variable to which write access is allowed. The iota quantifier of VDM-SL, denoted by ι, returns the unique value defined by its predicate part. The *mk-Coord* function is used to introduce the names x and y.

$Coord :: x, y : \mathbb{N}$

$Home\ ()$

ext wr $c : Coord$

post $c = (\iota\ Coord_{var} \in Coord \cdot Coord_{var} = mk\text{-}Coord(x, y) \wedge x = 0 \wedge y = 0)$

5.6.2 The calculation of pre- and post-conditions

The pre- and post-conditions which are calculated for the VDM-SL equivalents of the Z schemas are not the pre- and post-conditions which are described in (Hayes, 1987). There, the conditions which are calculated are the *implicit* pre- and post-conditions. Both of these are predicates over single states, obtained by existentially quantifying over the output and state-after variables of the schema predicate to obtain the pre-condition. The post-condition is obtained by existentially quantifying over the input and state-before variables of the schema predicate.

The post-condition thus generated is inappropriate because VDM post-conditions are predicates over two states. The VDM-SL translation of the pre-condition would be inappropriate if used as the pre-condition of the VDM-SL translation of the operation because it would be vacuous. The VDM equivalent is non-trivial and the Z-to-VDM-SL translator generates a non-trivial pre-condition. This means that a user of the translator will be required to undertake proofs that the generated pre-condition ensures that the post-condition is attainable for all possible starting states. Fortunately, this proof obligation is exactly Jones' implementability proof obligation for operations (Jones, 1990) and a disciplined Z author will have already discharged these proof obligations.

The pre- and post-conditions which are generated have the advantage of being simpler to calculate than the traditional Z equivalents. The schema predicate is partitioned into two sub-sequences. The first contains only those conjuncts of the predicate whose free variables are the state-before and input variables of the operation. The second contains the remaining conjuncts. The former is called the pre-condition: the latter the post-condition.

5.6.3 Schema expressions

The dialect of VDM-SL produced by the translator is based upon the language used in (Jones, 1990). This dialect has no direct equivalent of the Z schema calculus for combining schemas. Consequently, schema expressions are translated by expanding the expression using the definitions of the schema combinators to obtain a single schema. This is then translated as described above.

5.6.4 Given sets

A Z specification may contain a list of sets which the specification writer takes as 'given', i.e. they are assumed to exist but nothing else is assumed about them. Such sets have an equally formal status in the dialect of VDM-SL produced by the translator. A new keyword assume is introduced.

$[NAME, CHAR]$ translates as assume $NAME, CHAR;$

5.6.5 Axiomatic definitions

In Z, function and constant values with global scope are defined using an axiomatic definition. In VDM-SL, functions may be defined in an implicit pre .. post style but the purposes of the pre-condition and post-condition predicates are quite different from the purpose of the predicate used in a Z axiomatic definition. The pre-condition constrains the values of the arguments of the function; the post-condition constrains the values of the results of the function with respect to the arguments. Unlike Z axiomatic definitions, a VDM-SL function is not treated as a first-class object; there are no function composition operators in the language and a function f appears in predicates only in applied occurrences such as $f(x)$.

Axiomatic definitions are presently translated as function signatures followed by a post-condition restricting the behaviour of the function. The following Z function would be translated as shown below.

$$
\begin{array}{|l}
tens : \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\mathrm{ran}\ tens = \{x : \mathbb{N} \mid x \bmod 10 = 0\}
\end{array}
$$

$tens : \mathbb{N} \rightarrow \mathbb{N}$
post rng $tens = \{x \in \mathbb{N} \mid x \bmod 10 = 0\}$

5.7 Alternative designs for the translator

Alternative designs for a specification translator are now discussed and their relative merits are explored. One of the most obvious variations is the choice of input and output language. Another variation concerns the degree of user interaction with the translator.

5.7.1 Input and output languages

VDM-SL to Z: The difficulties of translation discussed in the previous section would not occur if the direction of translation were to be reversed. However, they would be replaced by alternative problems.

VDM-SL is founded on Scott domains whereas the semantics of Z is set-theoretic, and so recursive equations in Z must have a solution in set theory. Some VDM-SL descriptions of programming languages cannot be translated into Z: they require a domain-theoretic model. The following meaning function for a language which includes higher order functions is syntactically legal Z but has no model in set theory.

$[Function]$

$$| \; meaning\colon Function \rightarrow (Function \nrightarrow Function)$$

Another example of the limitations of set-theoretic languages is given in (Spivey, 1989). Consider a type containing the natural numbers as atoms and all functions from that type to itself.

$$T ::= atom\langle\langle \mathbb{N} \rangle\rangle \mid fun\langle\langle T \rightarrow T \rangle\rangle$$

No such set T can exist, because however large T is, there are many more functions from T to T than there are members of T. This means that there can be no injective function, *fun*, from $T \rightarrow T$ to T. This is ‘Cantor's Paradox’.

With reference to T, Spivey notes:

> The data type definition itself is used in the semantics of the lambda-calculus, and the need to make sense of it gave rise to the whole fruitful theory of Scott domains. In Z, we work not with domains but with sets, and there the definition is simply inconsistent.

Several less elevated problems present themselves. As noted on page 18, the direct definition style of function definition in VDM-SL does not have an exact equivalent in Z. Simple direct definitions would be obscured by being re-stated as definitions of variables constrained by a predicate. As an illustration, a simple function is defined directly and a possible translation is then stated. The function is the *max* function for natural numbers.

$$max : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$max(i, j) \quad \triangle \quad \mathsf{if}\ i \leqslant j\ \mathsf{then}\ j\ \mathsf{else}\ i$$

A possible Z translation is given below.

$$
\begin{array}{|l}
max : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
\hline
\forall i, j : \mathbb{N} \cdot \\
\qquad i \leqslant j \Rightarrow max(i, j) = j \land \\
\qquad i > j \Rightarrow max(i, j) = i
\end{array}
$$

Here, the difference in ease of comprehension of the two functions is tiny because of the extreme simplicity of the max function; but functions which use nested **if .. then .. else** constructs would produce considerably less intelligible results.

Differences in the naming conventions of the two languages also pose some problems. Z provides no equivalent of VDM-SL's projector functions for structured datatypes. A near equivalent of VDM-SL's '**compose**' and '::' constructors are the free types of Z. As an example, the following is a simple VDM-SL definition of a person.

$$
\begin{array}{ll}
Person :: & name : CHAR^*, \\
& age : \mathbb{N}
\end{array}
$$

The following is a suggested Z equivalent using the free type mechanism.

$$Person ::= mk\_Person \langle\!\langle \mathrm{seq}\ CHAR \times \mathbb{N} \rangle\!\rangle$$

With this translation, a mk_Person function is implicitly defined which enables the construction of variables of type $Person$ but, because the sub-fields of the datatype are not coupled to identifiers, the $name$ and age projector functions are not implicitly defined. Because the $name$ and age functions might be used widely throughout the VDM-SL specification, Z equivalents are desirable. A less compact definition would be required if the $name$ and age projector functions were required.

$$[Person]$$

$$
\begin{array}{|l}
mk\_Person : \mathrm{seq}\ CHAR \times \mathbb{N} \rightarrowtail Person \\
name : Person \to \mathrm{seq}\ CHAR \\
age : Person \to \mathbb{N} \\
\hline
\forall p : Person \cdot \\
\qquad mk\_Person(name(p), age(p)) = p
\end{array}
$$

Although these definitions are sufficiently compact to be acceptable, a VDM-SL datatype with more components would generate a correspondingly larger number of function definitions. Since datatypes such as $Person$ above are often used in VDM-SL specifications, a more compact translation would be desirable.

An alternative translation might be a schema; in this case a schema called $Person$ with components $name$ and age.

$$
\begin{array}{|l}
\hline
\textit{Person} \\
\hline
name : \mathrm{seq}\ CHAR; \\
age : \mathbb{N} \\
\hline
\end{array}
$$

Regrettably, although identifiers *name* and *age* are now available, there is no pre-defined equivalent of the *mk-Person* function that will be used in the VDM-SL specification. Use of Z's unique value μ function would be required more frequently than is desirable.

$$mk\text{-}Person(Ben, 2) \quad \text{translates as} \quad (\mu\, Person \mid name = Ben \land age = 2)$$

Some of the other possibilities for the source and target languages are now explained.

Z to Clear: Clear (Burstall & Goguen, 1980) would be an interesting choice for the output language of the translator. VDM-SL is sufficiently similar to Z to make it reasonably obvious that Z texts may be translated to VDM-SL. Clear is an algebraic specification language and, if chosen as the target language, would have permitted an interesting investigation of the differences and similarities between the model-oriented and property-oriented specification styles.

The increased difficulty of translation was discouraging. The verification burden of the chosen translator was considered to be sufficient. Additionally, it is desired to make this work as accessible to as many interested readers as possible. The knowledge that Clear is less widely known than VDM-SL made the choice of VDM-SL preferable in order to make this work more easily understood by a wider audience.

Z to programming language: Specifications translated into the dialects of VDM-SL and Clear considered as target languages above would not be executable in the conventional sense. A programming language such as Prolog (Clocksin & Mellish, 1987) might have been chosen. Z specifications could then have been exercised by execution as well as by proof.

Ignoring the added difficulty of the choice of a language such as Prolog as the target language, the ability to execute Z specifications was not considered desirable. Without interpreters for their specification languages, specification authors are forced to keep their specifications simple enough to be understood without needing to resort to executing the specification to discover what output a given input would produce.

In short, there are very many programming languages already and it is hoped that Z will not become a programming language.

5.7.2 Alternative user interfaces

The specification translator is non-interactive and non-incremental: it operates in batch mode and a specification must be translated in its entirety.

A translation assistant which interactively receives suggestions from the user about how to translate certain sections—or perhaps is non-interactive but recognizes translator directives in the specification text—would produce more readable VDM-SL output. That is, the output would more resemble a specification which had been written in VDM-SL initially by a skilled specification author.

However, an interactive Z to VDM-SL translation assistant would require the user to be skilled in both the Z and VDM specification styles and to have detailed knowledge of both languages. There are, of course, considerably fewer people fluent in both Z and VDM-SL than are fluent in Z alone. This fact, as much as the difficulty of verifying interactive or incremental software, makes the idea of an interactive Z to VDM-SL translator less appealing than the provision of a simpler non-interactive tool.

5.8 The implementation language used

The system is coded in Ada (Gehani, 1984; Barnes, 1989; McGettrick, 1982) throughout. The tasking facilities of the language are used. No implementation-dependent features of the language have been used.

The choice of Ada was prompted by several of the criteria listed on page 25. As an imperative language which provides explicit representation for parallelism, Ada allows efficient programs to be constructed. Production-quality compilers are also available. A module facility and a generic definition mechanism are provided, enabling the construction of the general, re-usable components required.

The strong typing of the language was an appealing feature since it provides detection of implementation errors at compile time. Variable initialization errors may be trapped by the compiler.

The Ada language is an interesting choice because it is not a 'verification language'. That is, it does not fit the conventional mould of languages used in verification projects where the languages favoured are small, semantically simple languages such as a 'clean' subset of Pascal (Jensen & Wirth, 1985). Ada is a massive, complex language which includes language constructs such as tasks and exceptions. The formal treatment of such constructs in software development is still an active research topic.

This concludes the overview of the specification translator. The following chapters present the modules of the translator in the order in which they were written. The order is essentially source-to-target order, beginning with the modules of the parser, then the core of the translator and the unparser.

Chapter 6

A Z Development of a Source Handler

A simple source handler is required to read a Z document, making available characters labelled with their row and column position. The source handler should be a reusable software unit, for use in lexical analyzers, syntax analyzers and other specification processing tools.

6.1 Specification

The source handler must recognize end of line and end of file characters. Upon detecting the end of line character, the current row position is incremented and the current column position reset. The end of file character is generated by the source handler once the Z document is exhausted. The end of line and end of file characters are distinct. A set of characters, $CHAR$, is assumed.

$$[CHAR]$$

$$\begin{array}{|l}
end_of_line, end_of_file : CHAR \\
\hline
end_of_line \neq end_of_file
\end{array}$$

The source handler accesses the Z document and maintains the current row and column position in the document. The Z document is a—possibly empty—sequence of characters. The row and column position are positive integers.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;State\;\hspace{20em}}} \\
document : \mathrm{seq}\ CHAR \\
row, column : \mathbb{N}_1 \\
\hline
\end{array}$$

The initial state of the source handler requires the row and column positions to be 1 and requires that the Z document does not contain the end_of_file character. This restriction is necessary since the end_of_file character is used as a special character in the specification and is understood to be recognized as a distinguished member of the $CHAR$ set.

$$State_0 \mathrel{\widehat=} [State \mid row = 1 \land column = 1 \land end\_of\_file \notin \operatorname{ran} document]$$

The source handler produces document characters which contain a character from the document and its document position. These are now formally described.

$$\begin{array}{|l}
\hline \textit{Document_Character} \\
ch : CHAR \\
row, column : \mathbb{N}_1 \\
\hline
\end{array}$$

6.1.1 Operation of the source handler

The source handler provides a single operation to make the next document character available. The three possible cases to be considered when supplying the next character relate to:

1. reaching the end of the document;
2. reaching the end of a line in the document; and
3. returning a single character.

In every case a character is provided whose document position is determined by the row and column values in the present state.

$$\begin{array}{|l}
\hline \textit{Establish_Position} \\
\Delta State \\
doc_char! : Document_Character \\
\hline
doc_char!.row = row \\
doc_char!.column = column \\
\hline
\end{array}$$

For the cases where the next document character is provided, it is useful to define an operation which makes this character available.

$$\begin{array}{|l}
\hline \textit{Obtain_Character} \\
\Delta State \\
doc_char! : Document_Character \\
\hline
document = \langle doc_char!.ch \rangle \frown document' \\
\hline
\end{array}$$

The actions taken for each case are now defined.

The *At_End_Of_File* operation

The following schema defines the action to be taken upon reaching the end of the document sequence. The end of file character is passed on. The state of the source handler does not change.

$$\begin{array}{|l}
\hline \textit{At_End_Of_File} \\
\Xi State \\
Establish_Position \\
\hline
document = \langle\rangle \\
doc_char!.ch = end_of_file \\
\hline
\end{array}$$

The *At_End_Of_Line* operation

At the end of a line of the Z input document, the source handler returns the end of line character with the corresponding document position. The end of line character is removed from the document, the current row number is incremented and the current column position reset to 1.

$$
\begin{array}{|l}
\textit{At\_End\_Of\_Line} \\
\hline
\textit{Establish\_Position} \\
\textit{Obtain\_Character} \\
\hline
\textit{head document} = \textit{end\_of\_line} \\
\textit{row}' = \textit{row} + 1 \\
\textit{column}' = 1 \\
\hline
\end{array}
$$

The *At_Next_Character* operation

If no end of line character has been detected and the document is not exhausted, the next character of the document is returned. It is removed from the beginning of the document and the column position is incremented.

$$
\begin{array}{|l}
\textit{At\_Next\_Character} \\
\hline
\textit{Establish\_Position} \\
\textit{Obtain\_Character} \\
\hline
\textit{head document} \neq \textit{end\_of\_line} \\
\textit{row}' = \textit{row} \\
\textit{column}' = \textit{column} + 1 \\
\hline
\end{array}
$$

The behaviour of the source handler is different for each of the three cases.

$$
\begin{array}{l}
\textit{Read\_Character} \mathrel{\widehat{=}} \\
\qquad \textit{At\_End\_Of\_File} \lor \textit{At\_End\_Of\_Line} \lor \textit{At\_Next\_Character}
\end{array}
$$

6.2 Development

The above specification can readily be refined to an implementation. The implementation will be sufficiently short and sufficiently easy to construct that little formal justification of the implementation will be required. Changes are detailed below.

6.2.1 Data refinement

The character set used will be the ASCII character set. A text file will be used to represent the sequence of characters. The *end_of_line* and *end_of_file* characters will be represented by the ASCII characters CR and SUB respectively. As required by the specification, these characters are distinct.

A large subrange of the positive integers will be used to represent the non-negative integers. Call this subrange *Document_Coordinate*. Define a *Document_Position* type to contain both a row and a column position.

$$Document\_Coordinate == 1 \mathrel{..} max\_coordinate$$

$$Document\_Position \mathrel{\widehat=} [row, column : Document\_Coordinate]$$

The state of the source handler is described by the following schema.

$$
\begin{array}{|l}
\hline
\textit{State} \\
document : \mathrm{seq}\ CHAR \\
row, column : Document\_Coordinate \\
\hline
\end{array}
$$

The document characters produced by the source handler are described below.

$$
\begin{array}{|l}
\hline
\textit{Document\_Character} \\
ch : CHAR \\
position : Document\_Position \\
\hline
\end{array}
$$

The row and column subcomponents of the document position part of a document character behave exactly as the row and column subcomponents of a document character described in the specification above.

6.2.2 Behaviour refinement

In this section, we consider the changes in the behaviour of the source handler necessitated by moving from the mathematical abstraction of the data to a programming language representation.

Consider first the possibility of failure when incrementing the row or column position of the document. In the respective cases, the row and column position are reset to 1 and a message sent to an error handler to indicate the overflow. Several other messages are also possible.

$$
\begin{array}{lcl}
Report ::= & & row\_number\_reset \\
& | & column\_number\_reset \\
& | & could\_not\_find \\
& | & could\_not\_open \\
& | & could\_not\_close \\
& | & processing\_complete
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Row\_Error} \\
\Delta State \\
message : Report \\
\hline
row + 1 \notin Document\_Coordinate \\
row' = 1 \\
message = row\_number\_reset \\
\hline
\end{array}
$$

The *Column_Error* schema is analogous.

Now the limitations of text files are considered and specifications are given for the actions to be taken upon being unable to find, open or close the file.

In each case, an error message is sent to the error handler. In some cases, the remedial action of generating an *end_of_file* character is also necessary.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Error_Action}} \\
\hline
\Xi State \\
doc_char!: Document_Character \\
\hline
doc_char!.ch = end_of_file \\
doc_char!.position.row = row \\
doc_char!.position.column = column \\
\hline
\end{array}$$

The messages sent for each of the three error conditions are now specified.

$$Search\_Failure \mathrel{\widehat=} \\ \quad Error\_Action \land [message: Report \mid message = could\_not\_find]$$

$$Open\_Failure \mathrel{\widehat=} \\ \quad Error\_Action \land [message: Report \mid message = could\_not\_open]$$

$$Close\_Failure \mathrel{\widehat=} \\ \quad Error\_Action \land [message: Report \mid message = could\_not\_close]$$

After completion—successful or otherwise—of the processing of the Z document by the source handler, the source handler must inform the error handler that no more diagnostic messages will be sent.

$$Finalize \mathrel{\widehat=} [message: Report \mid message = processing\_complete]$$

6.3 Implementation

The above development of the specification has provided a Z text which may be implemented directly in Ada. All of the error conditions related to file handling have direct counterparts in the exceptions provided by the standard Ada *Text_IO* package and may be removed from the main processing of the source handler in common with the structure of the above development.

No proof of satisfaction for the implementation is given. The development documented in the next chapter contains proofs of satisfaction for the implementation with respect to a Z development.

Chapter 7

A Z Development of a Lexical Token Generator

This chapter presents a correctness-oriented Z development of a lexical token generator which accepts document characters from the source handler and produces a stream of Z lexical tokens. The lexical tokens are composed of sequences of characters from the available character set, their classification and their document position. Sketch proofs of the implementation are provided.

7.1 Specification

The document characters accepted by the lexical token generator are described in the source handler specification (page 40) as follows.

$$\begin{array}{|l}
\textit{Document_Character} \\
\hline
ch : CHAR \\
position : Document_Position \\
\hline
\end{array}$$

Several sets can be formed which contain characters from particular categories within the character set.

$$\begin{array}{|l}
printable, non_printable, alphabet, \\
numerals, alphanumerics, punctuation : \mathbb{F}_1 CHAR \\
\hline
printable \cap non_printable = \varnothing \\
printable \cup non_printable = CHAR \\
alphabet \cap numerals = \varnothing \\
alphanumerics = alphabet \cup numerals \\
alphanumerics \subset printable \\
punctuation = printable \setminus alphanumerics
\end{array}$$

Certain distinct characters of the set are of particular interest. These are separators such as the space, tabulate and end of line characters; the single quotation mark character; the underscore character; and the end of file character.

It is significant that the six characters named above are distinct. If this were not the case, the specification would be inconsistent. When not using a formal notation to specify, small details such as the above are often left as silent assumptions. In a formal specification, the requirement that the six characters should be distinct is recorded in the predicate which constrains the possible values of the characters.

A formal statement of the properties and classifications of these characters now follows.

$$
\begin{array}{|l}
space, tab, quote, underscore, end\_of\_line, end\_of\_file : CHAR \\
\hline
\{space, quote, underscore\} \subset punctuation \\
\{tab, end\_of\_line, end\_of\_file\} \subset non\_printable \\
\#\{space, tab, quote, underscore, end\_of\_line, end\_of\_file\} = 6
\end{array}
$$

7.1.1 State of the lexical token generator

The lexical token generator accepts characters labelled with their document position from the source handler. A sequence of the document characters received is maintained by the token generator.

A Z document consists of sections of natural language text and mathematical text. The type of text being processed is also recorded by the token generator. The state of the token generator is now formally defined.

$$text\_type ::= natural\_language \mid mathematical$$

$$State \mathrel{\widehat{=}} [document : \operatorname{seq} Document\_Character;\ text : text\_type]$$

Initially, the document sequence is empty and the type of text expected is natural language text.

$$State_0 \mathrel{\widehat{=}} [State \mid document = \langle\rangle \land text = natural\_language]$$

7.1.2 Behaviour of the lexical token generator

The token generator accepts document characters from the source handler and produces lexical tokens. A lexical token consists of a sequence of characters along with its lexical classification and the document position of the first character in the sequence.

$$
\begin{array}{|l}
\hline
\textit{Lexical\_Token} \\
spelling : \operatorname{seq} CHAR \\
class : Lexical\_Classification \\
position : Document\_Position \\
\hline
\end{array}
$$

The categories of lexical classification used are defined in the next section. The operators for accepting a document character and supplying a lexical token are now developed.

Accepting a document character

The document character accepted is appended to the sequence of document characters already accepted. This causes no change in the type of text being processed by the token generator.

$$
\begin{array}{|l}
\hline
\textit{Accept\_Document\_Character} \\
\Delta State \\
doc\_char?: Document\_Character \\
\hline
document' = document \frown \langle doc\_char? \rangle \\
text' = text \\
\hline
\end{array}
$$

Sequence Operations

The first operation partitions the part of the document held by the token generator into two subsequences. The first of these is intended to be the next lexical token in the document. Additionally, two utility sequences of characters are gleaned from the two document parts. A *project_ch* function is required to obtain a character from a *Document_Character*. This can be generalised to apply to each element of a sequence of document characters using forward functional composition, denoted by a fat semi-colon, '$⨾$'.

$$
\begin{array}{|l}
project\_ch: Document\_Character \rightarrow CHAR \\
\hline
\forall dc: Document\_Character \cdot project\_ch(dc) = dc.ch
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Partition\_Document} \\
\Delta State \\
s_1, s_2: \mathrm{seq}\, Document\_Character \\
c_1, c_2: \mathrm{seq}\, CHAR \\
\hline
s_1 \frown s_2 = document \\
c_1 = s_1 ⨾ project\_ch \\
c_2 = s_2 ⨾ project\_ch \\
\hline
\end{array}
$$

In order to produce the next token of the document, the sequence s_1 above must be the longest meaningful initial subsequence of the document. The longest sequence is required in order that contiguous sequences of alphanumerics or digits can be interpreted as single alphanumeric names or numbers respectively. The sequence must be meaningful in the sense that it must be either a valid token, the initial part of a valid token or an unknown token consisting of not more than a single character.

The following operation obtains the longest meaningful initial subsequence of the document.

$$
\begin{array}{|l}
\hline
\textit{Obtain\_Sequence} \\
Partition\_Document \\
\hline
\forall k: \mathbb{N}_1 \cdot Z\_class\ (c_1 \frown ((1 \mathinner{..} k) \lhd c_2)) = Unknown \\
(\# c_1 = 1 \lor Z\_class\ \ c_1 \neq Unknown) \\
\hline
\end{array}
$$

The final sequence operation produces a lexical token from a sequence of document characters by stripping off the sequence of characters, classifying the sequence of document characters and determining the document position of the token.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Construct_Token \quad}} \\
Partition_Document \\
out!: Lexical_Token \\
\hline
out!.spelling = c_1 \\
out!.class = Natural_Language_Text \Rightarrow text = natural_language \\
out!.class = Z_class\ c_1 \Rightarrow text = mathematical \\
out!.position = (head\ document).position \\
\hline
\end{array}$$

Note that *position* is determined from the document sequence to ensure that it is defined even if s_1 is the empty sequence.

The *Input_Exhausted* Operation

Now consider the action of the token generator when the input is exhausted. The 'end-of-document' token is generated from the sequence. The state of the token generator remains unaltered.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Input_Exhausted \quad}} \\
\Xi State \\
Obtain_Sequence \\
Construct_Token \\
\hline
(head\ document).ch = end_of_file \\
\hline
\end{array}$$

The *Natural_Language_Token* Operation

This operation returns the longest natural language token at the beginning of the document sequence. It may be the empty sequence. Natural language text is a sequence of characters which does not contain the symbols 'Z', 'EZ' or the end-of-file character.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Natural_Language_Token \quad}} \\
Obtain_Sequence \\
Construct_Token \\
\hline
(head\ document).ch \neq end_of_file \\
text = natural_language \\
head\ c_2 \in \{Z, E, end_of_file\} \\
head\ c_2 = E \Rightarrow head\ (tail\ c_2) = Z \\
document' = s_2 \\
text' = mathematical \\
\hline
\end{array}$$

The *Z_Token* Operation

As with the *Natural_Language_Token* operation, this operation cannot be applied to an exhausted document. It may only be applied if the type of text being processed by the token generator is mathematical text. The next lexical token is removed from the document. If this token is the end-Z-section token 'EZ', the type of text to be processed next is natural language text, otherwise it remains mathematical text.

$$
\begin{array}{|l}
\hline
\textit{Z\_Token} \\
\textit{Obtain\_Sequence} \\
\textit{Construct\_Token} \\
\hline
(\textit{head document}).ch \neq \textit{end\_of\_file} \\
\textit{text} = \textit{mathematical} \\
\textit{document}' = s_2 \\
\textit{text}' = \textit{natural\_language} \Rightarrow \textit{out!.spelling} = \langle E, Z \rangle \\
\hline
\end{array}
$$

Producing a lexical token

In order to produce the next lexical token of the document, the token generator must be able to obtain as many characters as necessary from the source handler in order to form the token. This accessibility is ensured by demanding that the part of the document which has not been processed by the token generator be made available before a token is produced. Any implementation of this specification would only need to acquire sufficient document characters to determine the next token to be generated. The following schema ensures that the remainder of the document is available by insisting that the final character is the end of file character.

$$\textit{Remainder\_Available} \mathrel{\widehat=} [\Delta \textit{State} \mid (\textit{last document}).ch = \textit{end\_of\_file}]$$

Three distinct classes of token are produced by the token generator. These are natural language tokens, mathematical tokens and the distinguished token which indicates that the input document is exhausted. The operation of producing a lexical token is defined to be the act of generating a token from one of the three classes while having the remainder of the document available.

$$\begin{array}{l} \textit{Produce\_Token} \mathrel{\widehat=} \textit{Remainder\_Available} \wedge \\ \quad (\textit{Input\_Exhausted} \vee \textit{Natural\_Language\_Token} \vee \textit{Z\_Token}) \end{array}$$

The definitions of these three operations are given after the definitions of some utility sequence operations.

7.1.3 Classification functions

The possible classes of the lexical tokens are now defined. A Z document consists of sections of natural language text and sections of mathematical text. A lexical token is either a piece of natural language text, a Z token, the end of document token or a character sequence which cannot be classified as any of the above. The Z tokens recognised are numeric literals, sequences of alphanumerics with embedded underscores and sequences of single quotation mark characters used to decorate variable names in order to denote states after operations.

$$\begin{array}{rl} \textit{Lexical\_Classification} ::= & \textit{Natural\_Language\_Text} \\ \mid & \textit{Separator} \\ \mid & \textit{Numeric\_Literal} \\ \mid & \textit{Alphanumeric\_Name} \\ \mid & \textit{Decoration} \\ \mid & \textit{End\_Of\_Document} \\ \mid & \textit{Unknown} \end{array}$$

The following functions classify character sequences with one of the above classifications. The first of these describes which sequences of characters constitute separators between lexical tokens. A sequence of characters is classified as being a separator if it contains only space, tabulate or end of line characters.

$$\begin{array}{|l} separator : \mathrm{seq}\ CHAR \pfun Lexical\_Classification \\ \hline \forall s : \mathrm{seq}\ CHAR \cdot separator\ s = Separator \Rightarrow \\ \quad \mathrm{ran}\ s \subseteq \{space, tab, end\_of\_line\} \end{array}$$

The following function classifies numeric literals. A numeric literal is a sequence of characters from the numerals subset of the available characters.

$$\begin{array}{|l} numeric\_literal : \mathrm{seq}\ CHAR \pfun Lexical\_Classification \\ \hline \forall s : \mathrm{seq}\ CHAR \cdot numeric\_literal\ s = Numeric\_Literal \Rightarrow \\ \quad \mathrm{ran}\ s \subseteq numerals \end{array}$$

An alphanumeric name is a sequence of alphanumerics and underscores. The first must be a letter.

$$\begin{array}{|l} alphanumeric\_name : \mathrm{seq}\ CHAR \pfun Lexical\_Classification \\ \hline \forall s : \mathrm{seq}\ CHAR \cdot \\ \quad alphanumeric\_name\ s = Alphanumeric\_Name \Rightarrow \\ \quad\quad head\ s \in alphabet \land \\ \quad\quad \mathrm{ran}\ s \subseteq (alphanumerics \cup \{underscore\}) \end{array}$$

It is necessary to have a function to deal with the decorations which can be appended to an alphanumeric name.

$$\begin{array}{|l} decoration : \mathrm{seq}\ CHAR \pfun Lexical\_Classification \\ \hline \forall s : \mathrm{seq}\ CHAR \cdot decoration\ s = Decoration \Rightarrow \mathrm{ran}\ s = \{quote\} \end{array}$$

An end of document token is required to signal that the Z document file is exhausted. The sequence whose only character element is the end of file character is classified as the end of document token.

$$\begin{array}{|l} end\_of\_document : \mathrm{seq}\ CHAR \pfun Lexical\_Classification \\ \hline \forall s : \mathrm{seq}\ CHAR \cdot end\_of\_document\ s = End\_Of\_Document \Rightarrow \\ \quad s = \langle end\_of\_file \rangle \end{array}$$

Any sequences of characters which cannot be classified in any of the above categories are labelled unknown by the following function.

$$\begin{array}{|l} unknown : \mathrm{seq}\ CHAR \rightarrow Lexical\_Classification \\ \hline \mathrm{ran}\ unknown = \{Unknown\} \end{array}$$

The sequences of characters which are classified as natural language text are defined in a following section on the behaviour of the lexical token generator. The function which determines how to classify a sequence of characters from any other category is now defined.

$$\begin{array}{l} Z\_class == unknown \oplus end\_of\_document \oplus \\ \quad numeric\_literal \oplus decoration \oplus alphanumeric\_name \oplus separator \end{array}$$

7.2 Development

7.2.1 Data refinement

The state of the token generator is described below.

$$text\_type ::= natural\_language \mid mathematical$$

$$State \mathrel{\widehat=} [document : \mathrm{seq}\, Document\_Character;\ text : text\_type]$$

The two-valued descriptor of the type of text being processed may be represented by a value of a suitable enumerated type.

The token generator accepts document characters from the source handler and produces lexical tokens. A description of these follows.

Lexical_Token

$spelling : \mathrm{seq}\ CHAR$
$class : Lexical\_Classification$
$position : Document\_Position$

The only difficulty presented here—and in adequately representing the state of the token generator—is the representation of a sequence in a less elevated notation. The representation used contains a sequence of a fixed number of characters and an indicator of the significant part of the sequence.

This representation will require changes to the behaviour of the token generator. A larger number of possible classifications of a lexical token will be required. The amended description of the tokens produced by the token generator now follows.

$$String\_Length == 0 \mathrel{..} max\_string\_length$$

Lexical_Token

$spelling : \mathrm{seq}\ CHAR$
$length : String\_Length$
$class : Lexical\_Classification$
$position : Document\_Position$

$\# spelling = max\_string\_length$

Consider those lexical tokens which could have more than max_string_length characters, and which, as a result, require more than one fixed length sequence for their representation. These would be tokens from the classes : *Natural_Language_Text*; *Separator*; *Numeric_Literal*; *String_Literal*; *Alphanumeric_Name*; and *Decoration*.

Natural language text and separator tokens are split into a number of tokens of that class. The more structured tokens—numeric and string literals, alphanumeric names and decorations—are decomposed into a number of header tokens and a terminating token. The possible lexical classifications are now enumerated.

$$\begin{aligned}
Lexical_Classification ::= \; & Natural_Language_Text \\
\mid \; & Separator \\
\mid \; & Numeric_Header \\
\mid \; & Numeric_Literal \\
\mid \; & Alphanumeric_Header \\
\mid \; & Alphanumeric_Name \\
\mid \; & Decoration_Header \\
\mid \; & Decoration \\
\mid \; & End_Of_Document \\
\mid \; & Unknown
\end{aligned}$$

Justification of the representation chosen

Most of the above data refinements are self-evident. The choice of fixed length sequences provides a familiar blocked representation of a sequence. The original sequence may be retrieved by taking the concatenation of these sequences in the order in which they have been obtained from the token generator.

7.2.2 Behaviour refinement

The above design steps force further development of the behaviour of the token generator when accepting document characters and producing tokens.

Accepting a document character

The length of the sequence of document characters held by the token generator now has an upper bound. This restriction is incorporated in the specification of the operation which accepts document characters from the source handler.

$$\begin{array}{|l}
\hline
Accept_Document_Character \\
doc_char?: Document_Character \\
\Delta State \\
\hline
\#document < max_string_length \\
document' = document \frown \langle doc_char? \rangle \\
text' = text \\
\hline
\end{array}$$

Producing a document token

The operation of producing a token now produces a sequence of tokens whose spelling part is a sequence of not more than max_string_length characters.

Consider the problem of classifying the earlier parts of a segmented numeric literal, alphanumeric name or a name decoration. The initial parts of the sequence of tokens are classified as the appropriate header tokens. The schema which specifies this behaviour is given below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{Classify\_Token\_Parts}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
out!: \mathrm{seq}\, Lexical\_Token \\
\hline
(last\ out!).class = Numeric\_Literal \Rightarrow \\
\quad (\forall i: \mathrm{dom}(front\ out!) \cdot (out!(i)).class = Numeric\_Header) \\
(last\ out!).class = Alphanumeric\_Name \Rightarrow \\
\quad (\forall i: \mathrm{dom}(front\ out!) \cdot (out!(i)).class = Alphanumeric\_Header) \\
(last\ out!).class = Decoration \Rightarrow \\
\quad (\forall i: \mathrm{dom}(front\ out!) \cdot (out!(i)).class = Decoration\_Header) \\
\hline
\end{array}
$$

Constructing a sequence of lexical tokens involves partitioning the document and classifying the tokens. If the current text being processed is mathematical, the last lexical token is determined using the *Z_class* function. If the text currently being processed is natural language, the class of all the tokens is natural language.

$$
\begin{array}{|l}
project\_spelling: Lexical\_Token \rightarrow \mathrm{seq}\, CHAR \\
\hline
\forall l: Lexical\_Token \cdot project\_spelling(l) = l.spelling
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{Construct\_Token}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
Partition\_Document \\
Classify\_Token\_Parts \\
out!: \mathrm{seq}\, Lexical\_Token \\
\hline
{}^\frown/(out! \mathbin{;} project\_spelling) = c_1 \\
(last\ out!).class = Z\_class\ c_1 \Leftrightarrow text = mathematical \\
(head\ out!).position = (head\ document).position \\
\forall i: \mathrm{dom}\, out! \cdot \\
\quad (out!(i)).class = Natural\_Language\_Text \Leftrightarrow \\
\qquad text = natural\_language \\
\hline
\end{array}
$$

The *Z_Token* operation

This operation suffers a slight modification due to the change in the representation of the lexical tokens produced by the generator.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{Z\_Token}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
Obtain\_Sequence \\
Construct\_Token \\
\hline
(head\ document).ch \neq end\_of\_file \\
text = mathematical \\
document' = s_2 \\
text' = natural\_language \Rightarrow \#out! = 1 \wedge \\
\quad (head\ out!).length = 2 \wedge \\
\quad (1 \mathrel{..} 2) \lhd (head\ out!).spelling = \langle E, Z \rangle \\
\hline
\end{array}
$$

7.3 Implementation

This section provides a proof of an implementation of the lexical token generator. The proof is not formal: several sections of the code are taken as transparently correct.

Two simplifications are made. The first being the assumption that the correctness of any software used by the package has been established elsewhere. In addition, some simple transformations have been applied to the Z specifications in order to facilitate explanation.

This proof was not undertaken in tandem with the production of the software. It was attempted after the proposed implementation was complete in order to assess the correctness of an informally developed implementation. It is not suggested that delaying the production of the proof until after the implementation is complete is desirable. This strategy is only employed here because of the experimental nature of the development.

7.3.1 Preservation of a model of the state

The Z and Ada descriptions of the state of the token generator are compared below. The representations of parts of the state are expressed first in Z and then in Ada.

Output of the token generator

The function of the token generator is to produce lexical tokens. These consist of a fixed length sequence of characters along with a classification of the token and the document position of the first character in the sequence.

The Z description of these tokens is given below.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Lexical_Token}} \\
\hline
\textit{spelling}: \mathrm{seq}\ \textit{CHAR} \\
\textit{length}: \textit{String_Length} \\
\textit{class}: \textit{Lexical_Classification} \\
\textit{position}: \textit{Document_Position} \\
\hline
\#\textit{spelling} = \textit{max_string_length} \\
\hline
\end{array}$$

The Ada package contains the following type definitions. The Ada array literal '(others $\Rightarrow$ ' ')' denotes a one-dimensional character array with each location containing a space character.

```
type String_Length is range 0 .. Max_String_Length;
type String_Type is array (String_Length) of Character;
type Lexical_Token is
   record
      Spelling: String_Type := (others => ' ');
      Length: String_Length := 0;
      Class: Lexical_Classification;
      Position: Document_Position;
   end record;
```

There is certainly a superficial resemblance between the Ada datatype and the Z schema. The types of the sub-components of a lexical token are discussed in turn below.

Spelling: This is a sequence of exactly *max_string_length* characters. The Ada representation is an array of *max_string_length* + 1 characters, which is an adequate representation. The *CHAR* parameter of the Z specification is represented by the Ada standard type *Character* which is in turn the ASCII character set. This set contains representations of all the characters required by the specification.

type *String_Length* is range 0 . . *Max_String_Length*;
type *String_Type* is array (*String_Length*) of *Character*;

Length: The length of the string is defined above to be a natural number not greater than *max_string_length*. The Ada representation is a variable of an integer-derived type, *String_Length*.

Class: The Ada representation of the Z classifications of lexical tokens is an enumerated type.

Position: The representation of document position used is imported from the source handler package and is assumed to be an adequate representation.

Internal state of the token generator

The internal state of the token generator consists of a sequence of lexical tokens and a description of the type of text being processed.

$$text\_type ::= natural\_language \mid mathematical$$

$$State \mathrel{\widehat{=}} [document : \operatorname{seq} Document\_Character;\ text : text\_type]$$

The Ada representation economizes on storage by only storing two tokens at a time—it can be deduced from the specification that not more than two tokens are required in order to produce the next. This detail was not relevant at the specification stage. The type of text being processed is recorded by a variable of a predictable enumerated type.

Extensions to the state

The refinement may allow more states to be represented than could be modelled in the specification—it is still adequate. The following variables are also declared.

Current, *Next*: *Token_Index*;
Doc_Char: *Document_Character*;
Spare_Character: *Boolean*;
Previous_Token_Class: *Lexical_Classification*;

The higher dimensional Ada state-space offers the Z state-space as a projection. It is now appropriate to ask why the Ada state-space is larger when a smaller state would suffice. The reason is that the mathematical model strives for simplicity and clarity. In the implementation, redundant information is tolerated in order to avoid re-calculation of intermediate results by storing these results in the auxiliary variables declared above.

7.3.2 Achievement of the initial state

The token generator initially holds no tokens and expects to process natural language text.

$$State_0 == [State \mid document = \langle\rangle \wedge text = natural\_language]$$

The Ada variable initialisation mechanism provides a convenient way of ensuring that the initial state is achieved.

$$Text\colon Text\_Type := Natural\_Language;$$

7.3.3 Accepting a document character

The act of accepting a document character is described below.

$$\begin{array}{|l}
\hline
\textit{Accept_Document_Character} \\
\Delta State \\
doc_char?\colon Document_Character \\
\hline
\#document < max_string_length \\
document' = document \frown \langle doc_char? \rangle \\
text' = text \\
\hline
\end{array}$$

The pre-condition of the operation is shown below.

$$\#document < max\_string\_length$$

The Ada procedure which implements the concept of sequence concatenation in this context is shown below.

```
procedure Append(Token: in out Lexical_Token;
                 Char: in Document_Character) is
begin
   Token.Length := Token.Length + 1;
   Token.Spelling(Token.Length) := Char.Ch;
end Append;
```

This refinement may assume $\#document < max\_string\_length$. In Ada terms, this condition is $Token.Length < Max\_String\_Length$. Under this pre-condition, the procedure is guaranteed to terminate in a state which will satisfy the post-condition of the *Accept_Document_Character* operation.

7.3.4 Producing a lexical token

The Z schema *Produce_Token* is defined below to be the schema disjunction of the schemas implementing the delivery of the "end of document" token, a Z token or a natural language token under the assumption of the availability of as many document characters as are required.

$$Produce\_Token \mathrel{\widehat=} Remainder\_Available \land$$
$$(Input\_Exhausted \lor Natural\_Language\_Token \lor Z\_Token)$$

The production of the three kinds of lexical tokens is treated in later sections. Here, the pre-conditions of these operations are calculated. It is shown that suitable conditions are established by the Ada realization of the *Produce_Token* schema. The symbol '$\sqsubseteq$' is pronounced 'is refined by'.

PRE_INPUT_EXHAUSTED

$$(head\ document).ch = end\_of\_file \sqsubseteq Token(Current).Spelling(1) = EOF$$

PRE_NATURAL_LANGUAGE_TOKEN

$$(head\ document).ch \neq end\_of\_file \land text = natural\_language$$
$$\sqsubseteq$$
$$Token(Current).Spelling(1) \neq EOF \text{ and } Text = Natural\_Language$$

PRE_Z_TOKEN

$$(head\ document).ch \neq end\_of\_file \land text = mathematical$$
$$\sqsubseteq$$
$$Token(Current).Spelling(1) \neq EOF \text{ and } Text = Mathematical$$

Clearly, these conditions will be established by the *Produce_Token* operation shown below.

```
procedure Produce_Token is
   procedure Natural_Language_Token is ...;
   procedure Z_Token is ...;
begin
   if Token(Current).Spelling(1) = EOF then
      Token(Current).Class := End_Of_Document;
   elsif Text = Natural_Language then Natural_Language_Token;
   else Z_Token;
   end if;
   Previous_Token_Class := Token(Current).Class;
end Produce_Token;
```

The *Previous_Token_Class* variable records the class of the token returned as the result of the appropriate sub-procedure. The sub-procedures are now developed.

Producing the "End of document" token

From the *Input_Exhausted* schema, the procedure should obtain the longest meaningful initial subsequence of the document and construct a lexical token from this subsequence. Given that the first character of the document is the end of file character, the only allowable product would be the end-of-document token. The procedure returns such a token.

Natural language tokens

An incremental proof of this procedure is provided, showing that the conjuncts of the specification are satisfied by parts of the *Natural_Language_Token* procedure.

Specification error: At this point in the development a weakness was discovered in the specification of this operation. It should be noted that constructing a proof may be beneficial both for the specification and for the implementation.

A natural language section should be split into a sequence of tokens, each of which is classified as *Natural_Language_Text*. The initial version of the specification of the *Construct_Token* operation (page 51) required only the final token in the sequence to be classified in this way: the rest could have been given any classification. The predicate read:

$$(last\ out!).class = Natural\_Language\_Text \Leftrightarrow text = Natural\_Language$$

The version of the operation given on page 51 is the corrected version and contains the stronger requirement that all the tokens in the sequence be classified as given below.

$$\forall\, i : \mathrm{dom}\ out! \cdot \\ \quad out!(i).class = Natural\_Language\_Text \Leftrightarrow text = Natural\_Language$$

This requirement is met by the following assignment under the pre-condition of the operation.

$$Token(Current).Class := Natural\_Language\_Text;$$

After a sequence of natural language text tokens has been produced the lexical token generator is required to classify the text being processed as mathematical. This requirement will be satisfied by the following conditional statement in which *Z_Found*, *EZ_Found* and *End_Found* denote the detection of the relevant characters in the input. 'Z' and 'EZ' are brackets put around Z text meaning "Z text begins" and "End of Z text".

$$\mathsf{if}\ Z\_Found\ \mathsf{or}\ EZ\_Found\ \mathsf{or}\ End\_Found\ \mathsf{then}\ Text := Mathematical;\ \mathsf{end\ if};$$

The pre-condition of the operation ensures that the first character of the document is not the end of file character. It may be 'Z' or 'E'. The following conditional statement addresses this.

$$\begin{array}{l} \mathsf{if}\ Token(Current).Length = 1\ \mathsf{and}\ Token(Current).Spelling(1) = 'Z'\ \mathsf{then} \\ \quad Token(Next) := Token(Current); \\ \quad Token(Current).Length := 0; \\ \quad Z\_Found := True; \\ \mathsf{else}\ Construct\_Token; \\ \mathsf{end\ if}; \end{array}$$

The statement on the **then** limb produces an empty token and registers the detection of a Z-text-begins token. It remains only to show that the *Construct_Token* procedure builds a token detecting the three natural language text terminators.

Constructing a token

In the *Construct_Token* procedure, characters are repeatedly obtained from the source handler until 'Z', 'EZ' or the end of file character have been encountered. This is achieved by a simple iterative construct which maintains the invariant which is shown below.

The infix operator 'in' which is used in the statement of the invariant is a subsequence operator. Two sequences of characters are related by 'in' if the left operand is a contiguous subsequence of the right.

$$\begin{array}{l} \# document \leqslant max\_string\_length + 1 \wedge \\ \neg(\langle Z \rangle \text{ in } c_1) \wedge \\ \neg(\langle E, Z \rangle \text{ in } c_1) \wedge \\ \neg(\langle end\_of\_file \rangle \text{ in } c_1) \wedge \\ \# c_1 \leqslant max\_string\_length \end{array}$$

The loop is guaranteed to terminate because the number of document characters received is bounded from above and increases each time the loop is executed.

The following termination conditions may be deduced from the Z specification.

- $head\ c_2 = \langle Z \rangle$ or
- $(1 \mathinner{..} 2) \lhd c_2 = \langle E, Z \rangle$ or
- $head\ c_2 = end\_of\_file$ or
- $\# c_1 = max\_string\_length \wedge c_2 \neq \langle \rangle$.

Now express these conditions in Ada terms.

- $Token(Next).Spelling(1) = 'Z'$ or
- $Token(Next).Spelling(1 \mathinner{..} 2) = "EZ"$ or
- $Token(Next).Spelling(1) = EOF$ or
- $Token(Current).Length = Max\_String\_Length$ **and**
  $Token(Next).Length \neq 0$.

Now consider the behaviour of the token generator upon detecting a 'Z', 'EZ' or end-of-file character. The procedure appeals to a different sub-procedure in each case.

```
if Doc_Char.Ch = 'Z' then Check_For_Preceding_E;
elsif Doc_Char.Ch = 'E' then Check_For_Following_Z;
elsif Doc_Char.Ch = EOF then Process_End_Of_File;
else Append_If_Possible;
end if;
```

These sub-procedures are now developed in the order in which they occur in the conditional statement above.

Checking for a preceding letter 'E'

The current document character is a letter 'Z'. This procedure determines whether this represents a "Z text begins" symbol or is the second character of an "End of Z text" symbol. It establishes one of the *Z_Found* or *EZ_Found* boolean variables accordingly. One of the following conditions holds on exit.

- $Token(Next).Spelling(1) = 'Z'$ or
- $Token(Next).Spelling(1 \mathinner{..} 2) = "EZ"$.

Checking for a following letter 'Z'

The current document character is a letter 'E'. If it is followed by a letter 'Z', it is to be interpreted as an "End of Z text" token; if it is not, the characters are appended to the current token if possible, a new token being created otherwise. The procedure will establish the variables *EZ_Found* or *Token_Filled* as appropriate.

Processing the end of file character

A new token is created to hold the end-of-file character. The *End_Found* variable is assigned *True*.

```
procedure Process_End_Of_File is
begin
   New_Token(Token(Next), Doc_Char);
   End_Found := True;
end Process_End_Of_File;
```

Appending characters to the current token

The following procedure carries out the simple task of appending characters to the current token if this is possible. A new token is created otherwise.

```
procedure Append_If_Possible is
begin
   if Token(Current).Length = Max_String_Length then
      New_Token(Token(Next), Doc_Char);
      Token_Filled := True;
   else Append(Token(Current), Doc_Char);
   end if;
end Append_If_Possible;
```

Implementation error: At this point in the development an error was discovered in the Ada implementation. This was a simple substitution error introduced in producing a machine-readable version of the text. The handwritten version of the procedure was correct as is the version above. The statement on the else limb of the above conditional statement read: *Append(Token(Next), Doc_Char);*.

7.3.5 Producing Z tokens

As with the *Natural_Language_Token* operation, this operation cannot be applied to an exhausted document. It may only be applied if the type of text being processed by the token generator is mathematical text. The next lexical token is removed from the document. If this token is the end-Z-section token, the type of text to be processed next is natural language text, otherwise it remains mathematical text.

$$
\begin{array}{|l}
\hline
\textit{Z\_Token} \\
\textit{Obtain\_Sequence} \\
\textit{Construct\_Token} \\
\hline
(head\ document).ch \neq end\_of\_file \\
text = mathematical \\
document' = s_2 \\
text' = natural\_language \Rightarrow out!.spelling = \langle E, Z \rangle \\
\hline
\end{array}
$$

Under the precondition of the *Z_Token* operation, the *Text* variable of the Ada implementation is seen to record that the token generator is processing mathematical text when the *Z_Token* procedure is invoked. Thus, the following conditional statement is sufficient to meet the requirement of the operation that the variable recording the type of text being processed by the token generator is set to *Natural_Language* when the "end of Z text" token is encountered.

```
if Token(Current).Length = 2 and then Token(Current).Spelling(1 .. 2) = "EZ"
then Text := Natural_Language;
end if;
```

Constructing a token

The *Previous_Token_Class* variable records the classification of the previous token produced by the token generator. This variable is inspected initially. If the previous token was an initial part of any token, continue with a token of this class, otherwise inspect the first character of the token.

```
case Previous_Token_Class is
   when Numeric_Header => Continue_With(Numeric_Header, Numerals);
   when Alphanumeric_Header =>
      Continue_With(Alphanumeric_Header, Name_Characters);
   when Decoration_Header =>
      Continue_With(Decoration_Header, Decoration_String);
   when others => Inspect_First_Char;
end case;
```

Continuing with a token

Characters will be added until the current token contains at most *Max_String_Length* characters. The character should not be included if the *Ch_In_String* function returns *False*.

```
loop
   Next_Character;
   exit when Token(Current).Length = Max_String_Length or not Ch_In_String;
   Append(Token(Current), Doc_Char);
end loop;
New_Token(Token(Next), Doc_Char);
Classify_Token;
```

The *Ch_In_String* function implements a binary search for the character *Doc_Char.Ch* in the string which is supplied as a parameter to the *Continue_With* procedure. The implementation is straightforward and is omitted.

Classifying the token produced

In the *Classify_Token* sub-procedure the token produced will be either of the same class as the previous token or of a related class (e.g. *Numeric_Literal* for *Numeric_Header*). The *Lexical_Classification* type in the Ada text is ordered so that 'header' classifications are followed by the corresponding related class. Hence, the following conditional statement will be sufficient to classify the token. Here, C denotes the class of the previous token.

```
if C = Separator then Token(Current).Class := Separator;
elsif Ch_In_String then Token(Current).Class := C;
else Token(Current).Class := Lexical_Classification'Succ(C);
end if;
```

This concludes the section which deals with continuing with a token which is related to the previous one. If the token is not a continuation of the previously produced one, the first character of the token is inspected.

Inspecting the first character of a token

Unexpected characters are classified as *Unknown* and passed on.

```
case Token(Current).Spelling(1) is
   when Space | Tab | Eoln => Continue_With(Separator, Separator_Chars);
   when 'A' .. 'Z' | 'a' .. 'z' =>
      Continue_With(Alphanumeric_Header, Name_Characters);
   when '0' .. '9' => Continue_With(Numeric_Header, Numerals);
   when Quote => Continue_With(Decoration_Header, Decorations);
   when others =>
      Token(Current).Class := Unknown;
      Next_Character;
      New_Token( Token(Next), Doc_Char);
end case;
```

This completes the implementation of the lexical token generator.

7.4 Conclusions

What can one say about software for which a proof has been given at this level? The degree to which a reader would be convinced by such an argument would depend greatly on the quality of the author's prose. This is not a good correctness criterion. It would be incautious to declare the Ada text discussed here to be 'correct', but it has been reviewed by mathematical analysis. This analysis uncovered weaknesses in both the specification and the implementation and in consequence both were improved. No stronger claim can reasonably be made.

7.4.1 Disadvantages of the approach

What price has been paid for this increase in confidence? Some simple software engineering considerations concerning the finished implementation seem to have been overlooked. The 'Z' and 'EZ' tokens occur in place in the implementation text instead of having been declared as constants and given meaningful identifiers.

The Ada text—intentionally—contains only a comment at the beginning to explain where to find the specification, development and proof of the text. This is intended to encourage future maintainers of the implementation to read this development. However, integrating the specification and analysis with the implementation text would have been stronger encouragement.

7.4.2 Levels of formality

Software developers should always look for an appropriate level of proof for use in the various phases of a project. The following factors influence the level of proof:

1. the budget of the project;
2. the cost of failure;
3. the experience of the project team.

Chapter 8

A Z Development of a Terminal Generator

This chapter presents a development of a module of the specification translator which performs finer grain classification of tokens than that performed by the lexical token generator in the previous chapter. Refinement rules are devised for Z specifications and these rules are used to develop the specification to implementation.

8.1 Methodology

If formal methods of software development are to become generally useful for the disciplined production of large computer programs, it is believed that a reduction is required in the volume of proof obligations produced by the current methods.

The development method presented in this chapter complements operation decomposition by providing a less powerful but simpler decomposition. It is thought that it is most suitable for use in cases where operation decomposition seems a heavy hammer to use on a more pedestrian part of the problem being solved.

Removing the outer layer of the problem under study may throw the algorithmic parts of the problem into higher relief, thereby easing the selection of suitable algorithms for use in their solution. If existing rigorously—or formally—developed implementations of these algorithms exist, the implementor may choose to re-use this work rather than repeat it.

The process of solving a formal specification to find an implementation involves conveying objects from elegant and powerful languages to implementations in less powerful languages. This process may be eased merely by decomposing operations using the available operator calculus and by replacing operators by their definitions. It is advantageous to use a specification notation where some of the operators of the language are defined in terms of other operators of the language. Considerable progress may then be made without stepping outside the notation.

The process of using combinators to decompose operations advocates a use different from that suggested in (Spivey, 1988a) where the operation combinators are used to construct larger operations from smaller ones.

Here, the view taken is that programs are specifications. It is maintained that programming languages are only specification languages for which efficient interpreters have been—or may be—constructed. However, there are usually differences between conventional specifications and conventional programs which lead some authors to believe that programs are not specifications.

> 'Although some people have the tendency to blur the distinction between programs and specifications, we feel that there is a vital difference between supplying a BNF-like grammar and hand-coding a parser.' T. N. Nipkow in (Nipkow, 1986).

This statement does not reflect the view taken here. In this development the design steps taken are small and many and the transition from an initial, abstract specification to a polished, efficiently executable implementation is gradual.

8.2 Specification

The program fragment developed here accepts a sequence of lexical tokens from a lexical token generator and produces a sequence of terminal symbols of the Z concrete syntax as defined in (King *et al.*, 1988). It is called a 'terminal generator'.

The terminal generator is required to be a reusable tool suitable for use by syntax or type checkers, pretty-printers or other specification processing tools. Thus, it is required to deal with data which does not constitute a lexically correct Z document.

The environment of the terminal generator consists of its internal state, its input and its output. Each is specified in terms of schemas, giving initial and—unusually—final conditions for all.

State

The lexical tokens accepted by the terminal generator have the following form.

$$\begin{array}{|l}
\multicolumn{1}{l}{Lexical_Token} \\
\hline
spelling: \mathrm{seq}\ Character \\
length: String_Length \\
class: Lexical_Classification \\
position: Document_Position \\
\hline
\#spelling = max_string_length \\
\hline
\end{array}$$

There are ten possible lexical classifications, shown below.

$$\begin{array}{rl}
Lexical_Classification ::= & Natural_Language_Text \\
| & End_Of_Document \\
| & Separator \\
| & Decoration_Header \\
| & Decoration \\
| & Numeric_Header \\
| & Numeric_Literal \\
| & Alphanumeric_Header \\
| & Alphanumeric_Name \\
| & Unknown
\end{array}$$

The terminal symbols produced by the terminal generator have a similar form but are more precisely classified in terms of the terminal symbols of the Z concrete syntax.

$$
\begin{array}{|l}
\textit{Terminal\_Symbol} \\
\hline
\textit{spelling}: \mathrm{seq}\ \textit{Character} \\
\textit{length}: \textit{String\_Length} \\
\textit{class}: \textit{Terminal\_Symbol\_Class} \\
\textit{position}: \textit{Document\_Position} \\
\hline
\#\textit{spelling} = \textit{max\_string\_length}
\end{array}
$$

The terminal generator trades in lexical tokens and terminal symbols. It requires one variable of each kind.

In the initial state of the terminal generator, the *token* variable is used to record the token most recently accepted as input. It may not initially be the 'end of document' token.

The *terminal* variable is used in the construction of the current terminal symbol. The classification part of its prior value may also be useful in determining the symbol to be produced. For this reason, the classification part is given an initial value. This value is significant since it will influence the classification of the first token produced by the terminal generator. The exact classifications which the terminal generator will use have not yet been given, but two are *Identifier_Header* and *Selection*. The initial value may not be one of these. The terminal generator may terminate or refuse to supply terminals after producing the *End_Of_Text* terminal.

$$
\begin{array}{l}
\textit{State} \mathrel{\widehat{=}} \\
\quad [\textit{token}: \textit{Lexical\_Token};\ \textit{terminal}: \textit{Terminal\_Symbol}] \\[4pt]
\textit{Initial\_State} \mathrel{\widehat{=}} \\
\quad [\textit{State} \mid \textit{token.class} \neq \textit{End\_Of\_Document} \land \\
\qquad \textit{terminal.class} \notin \{\textit{Identifier\_Header}, \textit{Selection}\}] \\[4pt]
\textit{Final\_State} \mathrel{\widehat{=}} \\
\quad [\textit{State} \mid \textit{terminal.class} = \textit{End\_Of\_Text}]
\end{array}
$$

It might be thought that a suitable terminal generator could be produced quite easily. It might produce an 'end of document' terminal and then do nothing. In particular, it need not deal with any of the rest of this specification. Sadly, this would not suffice. The terminal generator must be brought from the initial state to the final state by a combination of the top level operations provided.

Input

The terminal generator accepts a sequence of lexical tokens as input. The input is required to contain only one 'end of document' token and this must be the last in the sequence. The final condition is more simple: the terminal generator is obliged to exhaust the input.

$$
\begin{array}{l}
\textit{Input} \mathrel{\widehat{=}} \\
\quad [\textit{in}?: \mathrm{seq}\ \textit{Lexical\_Token}]
\end{array}
$$

$$\begin{aligned}&\mathit{Initial\_Input} \mathrel{\widehat{=}}\\&\quad [\mathit{Input} \mid \exists_1\, i : \operatorname{dom} \mathit{in?} \cdot (\mathit{in?}\ i).\mathit{class} = \mathit{End\_Of\_Document} \land\\&\qquad (\mathit{last}\ \mathit{in?}).\mathit{class} = \mathit{End\_Of\_Document}]\end{aligned}$$

$$\begin{aligned}&\mathit{Final\_Input} \mathrel{\widehat{=}}\\&\quad [\mathit{Input} \mid \mathit{in?} = \langle\rangle]\end{aligned}$$

In the specification, the first element is removed from the input and held as the current token. An operation is now defined which achieves this. The operation may not change the value of the *terminal* variable.

$$\mathit{Read\_Token} \mathrel{\widehat{=}} [\Delta \mathit{State};\ \Delta \mathit{Input} \mid \mathit{in?} = \langle \mathit{token}' \rangle \frown \mathit{in?}' \land \mathit{terminal}' = \mathit{terminal}]$$

Output

As noted earlier, the terminal generator produces a sequence of terminal symbols. The output is a mirror image of the input. It is initially empty and must finally contain only one terminal which is classified as an 'end of document' terminal—the last.

$$\begin{aligned}&\mathit{Output} \mathrel{\widehat{=}}\\&\quad [\mathit{out!} : \operatorname{seq} \mathit{Terminal\_Symbol}]\end{aligned}$$

$$\begin{aligned}&\mathit{Initial\_Output} \mathrel{\widehat{=}}\\&\quad [\mathit{Output} \mid \mathit{out!} = \langle\rangle]\end{aligned}$$

$$\begin{aligned}&\mathit{Final\_Output} \mathrel{\widehat{=}}\\&\quad [\mathit{Output} \mid \exists_1\, i : \operatorname{dom} \mathit{out!} \cdot (\mathit{out!}\ i).\mathit{class} = \mathit{End\_Of\_Text} \land\\&\qquad (\mathit{last}\ \mathit{out!}).\mathit{class} = \mathit{End\_Of\_Text}]\end{aligned}$$

It is now possible to define the schema which appends a terminal symbol to the output.

$$\mathit{Output\_Terminal} \mathrel{\widehat{=}} [\Xi \mathit{State};\ \Delta \mathit{Output} \mid \mathit{out!}' = \mathit{out!} \frown \langle \mathit{terminal} \rangle]$$

8.2.1 The environment of the terminal generator

The environment of the terminal generator consists of its internal state, its input and its output. The environment is specified in terms of these schemas.

$$\mathit{Environment} \mathrel{\widehat{=}} \mathit{Input} \land \mathit{Output} \land \mathit{State}$$

$$\mathit{Initial\_Environment} \mathrel{\widehat{=}} \mathit{Initial\_Input} \land \mathit{Initial\_Output} \land \mathit{Initial\_State}$$

$$\mathit{Final\_Environment} \mathrel{\widehat{=}} \mathit{Final\_Input} \land \mathit{Final\_Output} \land \mathit{Final\_State}$$

8.2.2 The behaviour of the terminal generator

The production of terminals may be achieved by the sequential composition of three actions. Initially, a lexical token is accepted from the lexical token generator. Subsequently, any existing terminal symbol is processed. Finally, the next terminal is produced.

$$\mathit{Produce\_Terminal} \mathrel{\widehat{=}} \mathit{Read\_Token} \mathbin{;} \mathit{Process\_Existing\_Terminal} \mathbin{;} \mathit{Next\_Terminal}$$

Processing any existing terminal

The only circumstance in which an existing terminal symbol will have been withheld occurs upon receiving a lexical token from the token generator which may be interpreted as a selection symbol. Specify the operation to act only on selection tokens and to have no effect otherwise. The notation $S \oplus T$ is an abbreviation for $T \lor (\neg \text{ pre } T \land S)$ i.e. satisfy T if possible, S otherwise.

$$\begin{array}{l} \mathit{Process\_Existing\_Terminal} \mathrel{\widehat=} \\ \quad \Xi \mathit{Environment} \oplus [\mathit{Process\_Selection} \mid \mathit{terminal.class} = \mathit{Selection}] \end{array}$$

A selection symbol is a single dot, used to access a component of a structured object (e.g. the dot in *terminal.class*). This will have been classified as an unknown single punctuation character by the lexical token generator. The dot is also used in the number interval symbol (e.g. the two dots in $1 \mathinner{\ldotp\ldotp} 10$). Thus, the action to be taken depends on the current lexical token.

$$\mathit{Process\_Selection} \mathrel{\widehat=} \mathit{Token\_Is\_Not\_Selection} \oplus \mathit{Token\_Is\_Selection}$$

If the current lexical token also represents a selection terminal, classify both as a number interval, emit this and consume the next lexical token.

$$\begin{array}{|l} \hline \mathit{Produce\_Number\_Interval} \\ \Delta \mathit{State} \\ \hline \mathit{token.class} = \mathit{Unknown} \\ \mathit{Convert\_First}(\mathit{head}\ \mathit{token.spelling}) = \mathit{Selection} \\ \mathit{terminal'.position} = \mathit{terminal.position} \\ \mathit{terminal'.class} = \mathit{Number\_Interval} \\ \hline \end{array}$$

$$\begin{array}{l} \mathit{Token\_Is\_Selection} \mathrel{\widehat=} \\ \quad \mathit{Produce\_Number\_Interval} \mathbin{;} (\mathit{Output\_Terminal} \land \mathit{Read\_Token}) \end{array}$$

The *Convert_First* function is defined later. It is a total injective function from characters to terminal symbol classifications. If the next terminal is not a selection symbol, the withheld selection symbol is emitted.

$$\mathit{Token\_Is\_Not\_Selection} \mathrel{\widehat=} \mathit{Output\_Terminal}$$

Producing the next terminal

The treatment of the token received from the token generator is primarily influenced by its classification. Different schemas are defined for different token classes. The *Next_Terminal* schema is defined using schema overriding. By using the following ordering, a default classification of *Unknown* is obtained since the *Process_Unknown* schema is total—it has no execution pre-condition—whereas the others are partial.

$$\begin{array}{ll} \mathit{Next\_Terminal} \mathrel{\widehat=} & \mathit{Process\_Unknown} \oplus \\ & \mathit{Process\_End\_Of\_Document} \oplus \\ & \mathit{Process\_Separator} \oplus \\ & \mathit{Process\_Decoration} \oplus \\ & \mathit{Process\_Numeral} \oplus \\ & \mathit{Process\_Alphanumerics} \oplus \\ & \mathit{Process\_Natural\_Language\_Text} \end{array}$$

Before defining these operations, a useful schema which copies those parts of the input token which are needed in the current terminal symbol is defined. The document position is always copied. If the lexical representation of the terminal is significant, the *spelling* and *length* parts of the token are also copied. The lexical representation is significant if the token represents:

- explanatory prose;
- a natural number;
- an identifier;
- an unknown terminal.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Convert}} \\
\hline
\Delta State \\
\hline
terminal'.position = token.position \\
terminal'.class \in \{Explan_Text, Natural_Header, Natural_Number, \\
\quad Identifier_Header, Identifier, Unknown_Terminal\} \\
\quad\quad \Rightarrow terminal'.spelling = token.spelling \wedge \\
\quad\quad\quad terminal'.length = token.length \\
\hline
\end{array}$$

In many cases, the classification of the terminal may be decided from the class of the current lexical token alone. It is convenient to specify a version of the schema which performs this classification.

$$Convert\_On\_Class \mathrel{\widehat=} [Convert \mid terminal'.class = Convert\_Class(token.class)]$$

The *Convert_Class* function is a total function which converts lexical token classifications to terminal symbol classifications. It exhibits so little formulaic behaviour that it must be specified in extension.

$$\begin{array}{|l}
Convert_Class : Lexical_Classification \rightarrow Terminal_Symbol_Class \\
\hline
Convert_Class = \{Natural_Language_Text \mapsto Explan_Text, \\
\quad End_Of_Document \mapsto End_Of_Text, \\
\quad Separator \mapsto Explan_Text, \\
\quad Decoration_Header \mapsto Dashes, \\
\quad Decoration \mapsto Dashes, \\
\quad Numeric_Header \mapsto Natural_Header, \\
\quad Numeric_Literal \mapsto Natural_Number, \\
\quad Alphanumeric_Header \mapsto Identifier_Header, \\
\quad Alphanumeric_Name \mapsto Identifier, \\
\quad Unknown \mapsto Unknown_Terminal\}
\end{array}$$

With the aid of the *Convert_On_Class* schema, specification of some of the operations becomes quite simple. Notice that in the final case the 'separator' terminal is not passed on but it is recorded in order to overwrite the class of the previous token.

$$Process\_Document \mathrel{\widehat=} \\ \quad Convert\_On\_Class \mathbin{⨾} Output\_Terminal$$

$$Process\_End\_Of\_Document \mathrel{\widehat=} \\ \quad [Process\_Document \mid token.class = End\_Of\_Document]$$

$$Numeral \mathrel{\widehat=} \\ \quad Convert\_On\_Class \mathbin{⨾} Output\_Terminal$$

$$Process\_Numeral \mathrel{\widehat=} \\ \quad [Numeral \mid token.class \in \{Numeric\_Header, Numeric\_Literal\}]$$

$$Process\_Separator \mathrel{\widehat=} \\ \quad [Convert\_On\_Class \mid token.class = Separator]$$

The treatment of natural language text is only slightly more complex. The token generator may produce empty natural language text tokens which are to be removed by the terminal generator. Notice that it is important to register the empty natural language tokens in order to overwrite the class of the previous token.

$$Process\_Text \mathrel{\widehat=} \\ \quad Process\_Empty \oplus Process\_Significant$$

$$Process\_Natural\_Language\_Text \mathrel{\widehat=} \\ \quad [Process\_Text \mid token.class = Natural\_Language\_Text]$$

$$Significant \mathrel{\widehat=} \\ \quad Convert\_On\_Class \mathbin{⨾} Output\_Terminal$$

$$Process\_Significant \mathrel{\widehat=} \\ \quad [Significant \mid token.length \neq 0]$$

$$Process\_Empty \mathrel{\widehat=} \\ \quad Convert\_On\_Class$$

One of the more troublesome operations to specify is the decomposition of decoration sequences (e.g. the dashes in x''') into their component parts. A sequence of dashes of the same length as the decoration sequence is produced. The document position of each dash is calculated from the position of the sequence in the document and the position of the character within the sequence.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{Process_Decoration} \\
\Delta State \\
\Delta Output \\
\hline
\exists\, s : \mathrm{seq}\ Terminal_Symbol \mid \#s = token.length \cdot \\
\quad \forall\, i : \mathrm{dom}\ s \cdot \\
\quad\quad (s\ i).class = Dashes \land \\
\quad\quad (s\ i).position.row = token.position.row \land \\
\quad\quad (s\ i).position.column = token.position.column + i - 1 \land \\
\quad token.class \in \{Decoration_Header, Decoration\} \land \\
\quad terminal' = last\ s \land \\
\quad out!' = out! \frown s \\
\hline
\end{array}$$

The difficulty of distinguishing identifiers from reserved words may be eased by taking the decision to fix the length of the tokens generated by the token generator to that of the longest reserved word in the Z notation. Thus, when long alphanumerics are split into 'headers' and a following name, the headers may be immediately classified as identifier headers and the following name as an identifier.

$$
\begin{array}{l}
\mathit{Process\_Headers} \mathrel{\widehat=} \\
\quad [\mathit{Convert\_On\_Class} \mid \mathit{token.class} = \mathit{Alphanumeric\_Header}] \\
\mathit{Last\_Was\_Header} \mathrel{\widehat=} \\
\quad [\mathit{Convert\_On\_Class} \mid \mathit{terminal.class} = \mathit{Identifier\_Header}] \\
\mathit{Convert\_On\_Name} \mathrel{\widehat=} \\
\quad [\mathit{Convert} \mid \mathit{terminal}'.\mathit{class} = \\
\qquad \mathit{Convert\_Name}((1 \mathrel{..} \mathit{token.length}) \lhd \mathit{token.spelling})] \\
\mathit{Names} \mathrel{\widehat=} \\
\quad \mathit{Convert\_On\_Name} \oplus \mathit{Last\_Was\_Header} \\
\mathit{Process\_Names} \mathrel{\widehat=} \\
\quad [\mathit{Names} \mid \mathit{token.class} = \mathit{Alphanumeric\_Name}] \\
\mathit{Alphanumerics} \mathrel{\widehat=} \\
\quad \mathit{Process\_Headers} \lor \mathit{Process\_Names} \\
\mathit{Process\_Alphanumerics} \mathrel{\widehat=} \\
\quad \mathit{Alphanumerics} \mathbin{;} \mathit{Output\_Terminal}
\end{array}
$$

The $\mathit{Convert\_Name}$ function is a total function from character sequences to terminal symbol classes. It is composed of a partial terminal symbol table function and a 'default' total classification function.

$$\mathit{Convert\_Name} == \mathit{Class\_As\_Identifier} \oplus \mathit{Terminal\_Symbol\_Table}$$

The $\mathit{Class\_As\_Identifier}$ function classes all strings as identifiers.

$$
\begin{array}{|l}
\mathit{Class\_As\_Identifier} \colon \mathrm{seq}\ \mathit{Character} \rightarrow \mathit{Terminal\_Symbol\_Class} \\
\hline
\mathrm{ran}\ \mathit{Class\_As\_Identifier} = \{\mathit{Identifier}\}
\end{array}
$$

The terminal symbol table function is specified by enumeration of the pairings of strings and interpretations. The specification is lengthy and is presented in Appendix A.

It only remains to describe how to classify the symbols of unknown class produced by the token generator. These are all single character punctuation symbols. In some cases, these tokens will represent valid single characters of the Z language, e.g. '+' or '−'. In some cases, they may be invalid, e.g. '%' has no meaning in Z.

A $\mathit{Convert\_On\_First\_Char}$ schema is defined which is a specialization of the $\mathit{Convert}$ schema which determines the appropriate classification from the first character of the spelling of the token.

$$
\begin{array}{l}
\mathit{Convert\_On\_First\_Char} \mathrel{\widehat=} \\
\quad [\mathit{Convert} \mid \mathit{terminal}'.\mathit{class} = \mathit{Convert\_First}(\mathit{head}\ \mathit{token.spelling})]
\end{array}
$$

The *Convert_First* function—mentioned earlier—is a total function from characters to terminal symbol classes. It is composed of a total *Class_As_Unknown* function and a partial character classification function.

$$Convert\_First == Class\_As\_Unknown \oplus Character\_Class\_Table$$

$$\begin{array}{|l}
Class_As_Unknown : Character \rightarrow Terminal_Symbol_Class \\
\hline
\mathrm{ran}\ Class_As_Unknown = \{Unknown_Terminal\}
\end{array}$$

The character class table is a finite injective partial function from characters to terminal symbol classes.

$$\begin{array}{l}
Character_Class_Table == \\
\quad \{`;\text{'} \mapsto Semicolon, \\
\quad `,\text{'} \mapsto Comma, \\
\quad `!\text{'} \mapsto Exclam, \\
\quad `?\text{'} \mapsto Query, \\
\quad `=\text{'} \mapsto Equals, \\
\quad `:\text{'} \mapsto Colon, \\
\quad `_\text{'} \mapsto Place_Holder, \\
\quad `(\text{'} \mapsto Left_Parenthesis, \\
\quad `)\text{'} \mapsto Right_Parenthesis, \\
\quad `.\text{'} \mapsto Selection, \\
\quad `+\text{'} \mapsto Plus, \\
\quad `*\text{'} \mapsto Times, \\
\quad `-\text{'} \mapsto Minus, \\
\quad `>\text{'} \mapsto Greater_Than, \\
\quad `<\text{'} \mapsto Less_Than, \\
\quad `\#\text{'} \mapsto Cardinality\}
\end{array}$$

However, the *Process_Unknown* operation only surrenders the terminal which it has produced if it is not a 'selection' terminal, which may—as noted earlier—be the first character of a 'number interval' symbol.

$$\begin{array}{l}
Withhold_Selection \mathrel{\widehat=} \\
\quad [Convert_On_First_Char \mid terminal'.class = Selection] \\[1ex]
Emit_Non_Selection \mathrel{\widehat=} \\
\quad Convert_On_First_Char \fatsemi Output_Terminal \\[1ex]
Process_Unknown \mathrel{\widehat=} \\
\quad Emit_Non_Selection \oplus Withhold_Selection
\end{array}$$

The only remaining definition to supply is that of the *Terminal_Symbol_Class* set.

$$\begin{array}{l}
Terminal_Symbol_Class == \\
\quad \mathrm{ran}\ Convert_Class \cup \mathrm{ran}\ Convert_Name \cup \\
\qquad \mathrm{ran}\ Convert_First \cup \{Number_Interval\}
\end{array}$$

This completes the specification of the terminal generator.

8.3 Development

Three simple transformations are initially applied to the specification.

1. The definitions in the specification are placed in declare-before-use order. Such an ordering cannot always be found because mutually recursive definitions are permitted in Z. In the present development no recursive definitions are given.

2. The specification is brought close to a 'normal form' where only certain combinations of the Z schema calculus operators are used. In order to do this, new identifiers are generated from existing ones.

3. Schemas defined as a specialization of an existing schema are expanded to a simple schema definition if the conjoined predicate is a predicate over the final state. This transformation would benefit most from human intervention. In some cases, expansions could be avoided by re-arrangement and simplification of terms.

All three transformations are independent. They may be applied in any order or interleaved. It will be noted that all three could be fully automated if the first transformation is weakened from eliminating all forward references in the specification to minimizing the number of forward references. This concession approves specifications with mutually recursive definitions.

These transformations make the specification appear more program-like, but this says more about the amount of clerical detail in this specification than about the power of the transformations. In general, once the usual veneer of presentation is removed, formal specifications begin to resemble programs.

8.3.1 The calculation of pre-conditions

The calculation of the pre-conditions of the operations in the specification does not add any information to the specification and could be fully automated. However, the generated pre-conditions are subsequently rewritten and simplified. This is not an automatic process. The pre-conditions are simplified in order to clarify their meaning to the reader and, equally important, as a design step to reduce the power of the operators used in the statement of the pre-conditions. Typically this will involve removing the existential quantifiers which are introduced by the calculation of the pre-conditions.

Schemas declare some variables and state a relationship between them. The method used to calculate a pre-condition for a schema is described in (Hayes, 1987). The pre-condition is calculated by existentially quantifying over the output variables and removing them from the declarations—known as 'hiding'. That is, there must exist an output which can be related to the input by the schema predicate. This will always generate the weakest pre-condition—known as the *implicit* pre-condition.

For schemas specified in their familiar box form, this poses no problem. The output variables may be recognised syntactically since they end in an exclamation mark or are primed e.g. $out!$ or x'.

A problem arises with schemas which are specified by a schema expression or as a specialization of an existing schema. Here, it would be necessary to expand the schemas to obtain the descriptions in a form in which they can be manipulated. This would be possible, and could even be done automatically, but it would greatly decrease the clarity of the specification. Consequently, development does not proceed by this route—especially since the schema combinators are often used in the current case study. Instead, a structured way of calculating the pre-conditions is found by deriving rules which define how the schema combinators interact with the precondition operator.

Caveat: The rules given in this chapter are not general refinement rules for Z. The rules were devised for this development alone after the initial specification was complete. They always produce valid refinements for this development but would not always produce satisfactory refinements when applied to other specifications.

For example, a later rule allows the sequential composition operator of Z to be replaced by sequential composition in Ada. (Spivey, 1989) gives a case where this rule produces an incorrect refinement. If Op_1 and Op_2 are non-deterministic, $Op_1 \mathbin{⨾} Op_2$ cannot always be refined to Op_1; Op_2. In $Op_1 \mathbin{⨾} Op_2$, the state in which Op_1 finishes is chosen to satisfy the pre-condition of Op_2. Thus, the precondition of $Op_1 \mathbin{⨾} Op_2$ requires only that an intermediate state exists. None of the schema expressions in this specification relies on this property. This allows the simpler rule to be used.

Because such subtleties must be considered, universally applicable refinement rules for Z would be significantly more difficult to devise than the specialized rules used here and would almost certainly be more complex.

8.3.2 Derived rules of the schema calculus

The following rules say how the pre-condition operator interacts with the schema combinators used in the above specification.

Synonym rule

The first rule simply states that 'two syntactically equivalent schemas have syntactically equivalent pre-conditions'.

$$\frac{R \mathrel{\widehat=} S}{\text{pre } R \mathrel{\widehat=} \text{pre } S}$$

Restriction rule

This rule states that 'if R is formed from S by conjoining to its predicate part a predicate over the initial state then the pre-condition of R may be obtained by conjoining the predicate to the pre-condition of S'.

$$\frac{R \mathrel{\widehat=} [S \mid p]}{\text{pre } R \mathrel{\widehat=} [\text{pre } S \mid p]} \quad [\ \mathit{initial}(p)\ ]$$

Unfortunately, this rule will not always hold if the predicate is allowed to involve variables of the final state. A similar difficulty is encountered when describing schema conjunctions.

Rule of conjunction

It is impossible to give a general rule which describes the pre-condition of $S \wedge T$ solely in terms of the pre-conditions of S and T. However, as with the restriction rule, it is possible to give a rule which may be applied in certain circumstances. This rule states that 'if the state-spaces of S and T are disjoint, then the pre-condition of the conjunction is the conjunction of the pre-conditions'.

$$\frac{R \mathrel{\widehat=} S \wedge T}{\mathrm{pre}\ R \mathrel{\widehat=} \mathrm{pre}\ S \wedge \mathrm{pre}\ T} \quad [\, \Theta S \cap \Theta T = \varnothing \,]$$

Rule of disjunction

This rule states that the pre-condition operator distributes through the disjunction operator.

$$\frac{R \mathrel{\widehat=} S \vee T}{\mathrm{pre}\ R \mathrel{\widehat=} \mathrm{pre}\ S \vee \mathrm{pre}\ T}$$

Rule of overriding

This rule states that 'if R is formed by overriding schema S by schema T, the pre-condition of R may be obtained by disjoining the pre-conditions of T and S'.

$$\frac{R \mathrel{\widehat=} S \oplus T}{\mathrm{pre}\ R \mathrel{\widehat=} \mathrm{pre}\ S \vee \mathrm{pre}\ T}$$

Rule of composition

This rule states that 'if R is formed by composing S and T, its pre-condition may be obtained by composing S with the pre-condition of T and finding the pre-condition of the result'.

$$\frac{R \mathrel{\widehat=} S \mathbin{;} T}{\mathrm{pre}\ R \mathrel{\widehat=} \mathrm{pre}\ (S \mathbin{;} \mathrm{pre}\ T)}$$

A specification in normal form is one all of whose derived operations are expressed in a form corresponding to one of the above rules.

The pre-conditions of the operations in the specification may now be calculated. The arrangement of the operations in declare-before-use order enables optimal use of the pre-conditions of the earlier operations in calculating the pre-conditions of later ones.

Since the pre-conditions above were calculated from the specification they may be merged with the operations without introducing inconsistencies. This activity may also be fully automated.

When the pre-conditions and the operations are merged, the pre-condition information in the specifications of the operations becomes redundant. This is removed wherever this does not mar the specification. The 'post' operator of Z does not return such a schema post-condition, it returns a schema with a predicate over a single state. Posit a new post-condition operator, Φ, which removes the redundant information, giving a suitably reduced version of a schema. The behaviour of Φ is restricted by the following equivalence.

$$T \mathrel{\widehat{=}} \text{pre } T \land \Phi T$$

The above will have a solution for all schemas: T itself is a solution. However, ΦT denotes a schema which corresponds to a VDM post-condition for the operation T, that is, the predicate part is a predicate over the initial and final states which makes no redundant demands on the initial state.

This is also a convenient time to repair some of the damage done to the clarity of the specification by the earlier expansion of some schema definitions. This may be thought of as a form of the classical program transformation technique known as 'folding'. Folding an expression replaces it wherever it occurs by the name of an equivalent definition. When the current definition is replaced by an earlier definition, no inconsistency is introduced because calculating consequences of a specification cannot alter its meaning. Consequently, where there are several alternative descriptions of an operation, any one may be chosen.

The calculation of post-conditions of schemas where an operator of the schema calculus is used in the definition is achieved by rules similar to those above for calculating pre-conditions.

8.3.3 Translation rules

The following rules suggest Ada representations of Z schema calculus constructs. They are rather less exacting and rather more specialized than those in (Morgan, 1990) where the refinement calculus given suggests refinements of specification statements in Dijkstra's non-deterministic do .. od language.

The rules should not be thought to be defining the meaning of the Ada constructs involved as do the axiomatic proof rules given in (McGettrick, 1982). Instead, they suggest possible refinements in the Ada notation. These will often be more deterministic than the corresponding Z expressions.

Although many of the transformations are still purely syntactic, some are *semantic transformations*. These are described in (Broy, 1983) as those transformations where the applicability condition contains semantic information and is generally not decidable. These often require human intervention in their application.

Pre-condition expression rule

This rule states that an expression in a pre-condition may be expressed in Ada using the pre-defined equality operator.

$$\frac{v = e}{v_c = e_c} \quad [\ equality\_defined(v_c)\ ]$$

Here, v_c and e_c represent refinements of the variable and the expression respectively.

This rule is not universally applicable. There are some types in Ada for which an equality test is not provided e.g. files.

Post-condition expression rule

This rule states that an expression in a post-condition may be expressed in Ada using the assignment operator.

$$\frac{v' = e'}{v_c := e_c} \quad [\, assignment\_defined(v_c) \,]$$

This rule is also not universally applicable. There are some types in Ada for which assignment is not defined e.g. files. It may be useful to give a special form of this rule by which redundant assignments can be removed.

$$\frac{v' = v}{\textsf{null};}$$

Schema rule

This rule states that a schema definition may be represented in Ada by a procedure definition.

$$\frac{R \mathrel{\widehat=} e}{\textsf{procedure } R \textsf{ is begin } e_c \textsf{ end };}$$

Here, e_c represents the schema expression e on the concrete level. A version could also be given which includes a '$params_c$' which represents the parameters to the operation R on the concrete level. A special form of the rule can be given for cases where the syntactic equality is used to define a synonym for an operation.

$$\frac{R \mathrel{\widehat=} S}{\textsf{procedure } R \textsf{ renames } S;}$$

This transformation is not universally applicable. A procedure defined by a renames clause is classified as a basic declaration in Ada and must appear in a basic declaration section.

Schema pre-condition rule

This rule states that a schema pre-condition may be represented in Ada by a Boolean function definition.

$$\frac{\mathrm{pre}\ R \mathrel{\widehat=} e}{\textsf{function } PRE\_R \textsf{ return } BOOLEAN \textsf{ is begin return } e_c \textsf{ end };}$$

Here, e_c represents the expression e on the concrete level. A special form of the rule can be given for cases where the syntactic equality is used to define two pre-conditions to be synonymous.

$$\frac{\mathrm{pre}\ R \mathrel{\widehat=} \mathrm{pre}\ S}{\textsf{function } PRE\_R \textsf{ return } BOOLEAN \textsf{ renames } PRE\_S;}$$

Rule of conjunction

This rule suggests a refinement of a schema conjunction as the sequential composition of refinements of the operations. This rule may only be applied if the state-spaces of the two operations are disjoint.

$$\frac{S \wedge T}{S_c\, T_c} \quad [\, \Theta S \cap \Theta T = \varnothing \,]$$

Recall that Ada statements are sequentially composed by writing them one after the other. The semicolon is a statement terminator in Ada.

Rule of disjunction

This rule suggests the refinement of a schema disjunction as a conditional statement which if the refinement of the pre-condition of the second operation is satisfied then the refinement of the second operation is performed. If not, the refinement of the first operation is performed.

$$\frac{S \vee T}{\mathsf{if}\ (\mathrm{pre}\ T)_c\ \mathsf{then}\ T_c\ \mathsf{else}\ S_c\ \mathsf{end\ if;}}$$

Rule of overriding

The rule for schema overriding is similar to the above.

$$\frac{S \oplus T}{\mathsf{if}\ (\mathrm{pre}\ T)_c\ \mathsf{then}\ T_c\ \mathsf{else}\ S_c\ \mathsf{end\ if;}}$$

For expressions involving several overriding operators, the conditional statement contains an elsif limb.

$$\frac{R \oplus S \oplus T}{\mathsf{if}\ (\mathrm{pre}\ T)_c\ \mathsf{then}\ T_c\ \mathsf{elsif}\ (\mathrm{pre}\ S)_c\ \mathsf{then}\ S_c\ \mathsf{else}\ R_c\ \mathsf{end\ if;}}$$

Translation can in large part be achieved by a form of the classical program transformation technique known as unfolding. Unfolding simply replaces a name by its definition. The schema overriding operator is a good candidate for unfolding because a compact definition of this operator can be given.

$$S \oplus T \mathrel{\widehat{=}} (\mathrm{pre}\ T \wedge \Phi T) \vee (\neg \mathrm{pre}\ T \wedge S)$$

Here, as before, ΦT may be any schema which when conjoined with pre T gives T. This form of the definition is preferred because the expression pre T appears twice. This indicates that this subexpression may be factored out of the expression and evaluated only once in the final implementation.

Rule of composition

In the Ada representation of schema composition, refine both operations and elide the semicolon. As noted in the caveat on page 73, this rule fails to produce correct refinements under certain conditions. Such conditions do not occur in the specification in this chapter.

$$\frac{S \fatsemi T}{S_c T_c}$$

8.4 Implementation

When applied to the specification, the above transformations yield an Ada implementation with relative ease. The implementation contains many small, simple Boolean-valued functions such as the one below.

```
function pre_Process_End_Of_Document return Boolean is
begin
   return Token.Class = End_Of_Document;
end pre_Process_End_Of_Document;
```

Some of the more difficult refinements are now discussed.

In Z *external* aspects are treated by designating certain variables as containing initially the *input* values, and certain others as containing the *output* values. This means that the specifier is obliged to think of file input as accepting the entire file initially, holding it internally, and removing items from the front as required. The treatment of output is similar. This abnormality becomes troublesome when refining operations involving input or output. The signatures of the Ada implementations of the operations involving input and output differ from the signatures of the corresponding Z operations and there is no direct representation of the input sequence or the output sequence in this package of the translator.

A relatively difficult refinement is that of the *Process_Decoration* operation (page 69). The specification of this operation uses a universal quantifier to define the elements of a sequence s of length *token.length*. The operation is refined by using the *Terminal* variable *Token.Length* times and replacing the quantifier by a for loop.

```
procedure Process_Decoration is
begin
   for I in 1 .. Token.Length loop
      Terminal.Class := Dashes;
      Terminal.Position.Row := Token.Position.Row;
      Terminal.Position.Column :=
         Token.Position.Column + Document_Coordinate(I) − 1;
      Output_Terminal;
   end loop;
end Process_Decoration;
```

The initial Ada implementation produced by the transformations described above is subsequently clarified by applying classical transformation techniques. Procedures and functions are unfolded; redundancies and synonyms are removed; and subprograms are nested to amplify the structure of the refinement process. The nesting of subprograms is needed because the implementation retains the flatness of the specification. Z is not a block-structured language and subspecifications cannot be nested.

8.5 Conclusions

8.5.1 Advantages of the method

The development method used here does not generate as many proof obligations as the rigorous program development techniques.

> 'Reducing guesswork and proof to calculation is how mathematicians simplify their own tasks, as well as those of users of mathematics—the scientist, the engineer and now the programmer.' C.A.R. Hoare in (Hoare, 1986)

A useful advantage is a reduction in the amount of memory work required on the part of the programmer. Eric Hehner in (Hehner, 1984) points out that programmers attempt to remember what they are assuming about each of the program variables at any point in the program. Here, no such memory feat seems to be required—or at worst a more modest one. An added advantage is the documentation of the thought process at work in program creation which would be useful if attempting adaptive program maintenance.

It is suggested that with an appropriate software tool to support this style of programming, the implementation given here could have been rigorously developed in a time which would be considered acceptable in an industrial software environment.

8.5.2 Disadvantages of the method

A side-effect of any manual program creation is a detailed human study of the—formal or informal—specification of the problem. This can lead to a deeper understanding of the relationship between the specification and the informal statement of the customer's requirements—this activity is termed 'validation'. Machines cannot validate. If any part of the programming activity is automated, the validation which accompanies it may not be addressed. Consequently, incongruities between the specification and the requirements which might have been found by manual methods may go undetected.

Chapter 9

A VDM Development of a Z Parser

This chapter presents a summary of a VDM specification which has been type-checked by the QUB VDM toolset (McParland, 1989). The Z parser first forms terminal symbols of the concrete syntax of Z from lower level terminals and from these forms a parse tree. The specification reflects this division by defining a 'pre-parser' which performs the first task and defining the parser core by an EBNF syntax given in Appendix B.

The Z parser described accepts checked Z texts. As such, it is not enforcing the language definition: merely accepting sentences of the language and constructing a parse tree. The parser does not check for errors in the input text. This considerably simplifies the specification and subsequent implementation.

9.1 Specification

9.1.1 Types

This section describes the types which are defined in the specification. The following types may be derived from the syntax given in Appendix B and they are not repeated here.

- The type *Concrete-Terminal-Class* which is the type formed by taking the union of the types *relation*, *prefix-function*, *postfix-function*, *operator*, *right-operator* and *concrete-terminal*; these constituent types are also omitted.

- The type *Terminal-Symbol-Class* which unites the *Concrete-Terminal-Class* and *Basename-Parts* types.

The terminal symbols which are accepted by the pre-parser contain a spelling part, a classification of this spelling and a document position part. The definitions of the types *String* and *Document-Position* are not given here.

$$\begin{array}{lll} \textit{Terminal-Symbol} & :: & \textit{spelling}\colon \textit{String} \\ & & \textit{class}\colon \textit{Terminal-Symbol-Class} \\ & & \textit{position}\colon \textit{Document-Position} \end{array}$$

Three kinds of terminals are produced; basename terminals, natural language text and terminals of the concrete syntax. Although natural language text and 'basenames', or 'identifiers' as they are more commonly known, are included in the concrete syntax of Z, they are only defined informally. Thus this separation into three types follows naturally from the concrete syntax.

$$\mathit{Terminal} = \mathit{Natural\text{-}Language\text{-}Text} \cup \mathit{Concrete\text{-}Terminal\text{-}Symbol} \cup \mathit{Basename}$$

9.1.2 Input and output

The pre-parser accepts a sequence of terminal symbols as input and produces a sequence of terminals. The output is initially empty.

$$\mathit{init\text{-}output} = [\,]$$

9.1.3 Natural language text

The processing of natural language text simply involves packaging the text as a terminal symbol and then issuing this symbol.

$$\begin{array}{lll} \mathit{Natural\text{-}Language\text{-}Text} & :: & \mathit{text}\colon \mathit{String} \\ & & \mathit{position}\colon \mathit{Document\text{-}Position} \end{array}$$

Notation: The following operation uses the let .. in construct of VDM-SL. This construct is used to introduce a declaration which is local to an expression. A similar construct is provided by some functional programming languages and also by higher-order procedural languages such as ML (Wikström, 1987). In such languages, it is used to improve the efficiency of execution of a function by factoring out common sub-expressions which need only be evaluated once. In VDM-SL it is used only to introduce an abbreviation for an expression. It is often used in tandem with the application of a *mk*-function to introduce an abbreviation for application of one of the projector functions. For example, given $\mathit{Coord}::x, y\colon \mathbb{R}$ and $c\colon \mathit{Coord}$, then let $\mathit{mk\text{-}Coord}(x, y) = c$ in exp introduces the names x and y as abbreviations for $x(c)$ and $y(c)$ respectively.

$$\mathit{PROCESS\text{-}NATURAL\text{-}LANGUAGE\text{-}TEXT}\ ()$$

$$\begin{array}{l}
\mathsf{ext\ wr}\ \mathit{input}\colon \mathit{Terminal\text{-}Symbol}^* \\
\quad \mathsf{wr}\ \mathit{output}\colon \mathit{Terminal}^* \\
\mathsf{pre}\ \mathit{class}(\mathsf{hd}\ \mathit{input}) = \mathit{explan\text{-}text} \\
\mathsf{post\ let}\ \mathit{mk\text{-}Terminal\text{-}Symbol}(s, c, p) = \mathsf{hd}\ \overleftarrow{\mathit{input}}\ \mathsf{in} \\
\qquad \mathit{input} = \mathsf{tl}\ \overleftarrow{\mathit{input}} \wedge \\
\qquad \mathit{output} = \overleftarrow{\mathit{output}} \curvearrowright [\mathit{mk\text{-}Natural\text{-}Language\text{-}Text}(s, p)]
\end{array}$$

9.1.4 Concrete terminals

Concrete terminals are given a more general classification in addition to their classification as reserved words or punctuation symbols of the language.

$$\begin{array}{lll} \mathit{Concrete\text{-}Terminal\text{-}Symbol} & :: & \mathit{class}\colon \mathit{Z\text{-}Terminal\text{-}Class} \\ & & \mathit{terminal}\colon \mathit{Concrete\text{-}Terminal\text{-}Class} \\ & & \mathit{position}\colon \mathit{Document\text{-}Position} \end{array}$$

where

$$inv\text{-}Concrete\text{-}Terminal\text{-}Symbol(mk\text{-}Concrete\text{-}Terminal\text{-}Symbol(c, t, p)) \quad \triangleq$$
$$\quad t \neq explan\text{-}text$$

Seven more general classifications are now defined. These will simplify the task of parsing by allowing the parser to process expressions which use, for example, prefix or postfix functions without concern about which prefix or postfix function is being applied. Such semantic information will not be relevant until the translation phase, where it is necessary to distinguish between functions of the same class such as *head* and *tail*.

$$\begin{array}{lcl} Z\text{-}Terminal\text{-}Class & = & \text{Z-RELATION} \\ & \mid & \text{Z-PREFIX-FUNCTION} \\ & \mid & \text{Z-POSTFIX-FUNCTION} \\ & \mid & \text{Z-RIGHT-OPERATOR} \\ & \mid & \text{Z-OPERATOR} \\ & \mid & \text{Z-CONCRETE-TERMINAL} \\ & \mid & \text{Z-BASENAME} \end{array}$$

Processing concrete terminal symbols is only slightly more complex than processing natural language text. An auxiliary *classify* function is used to tag the terminal symbols with the appropriate higher-level classification.

$$classify : Terminal\text{-}Symbol\text{-}Class \to Z\text{-}Terminal\text{-}Class$$

$$\begin{array}{lll} classify(c) & \triangleq & \textsf{cases } c \textsf{ of} \\ & & \begin{array}{ll} mk\text{-}relation(x) & \to \text{Z-RELATION} \\ mk\text{-}prefix\text{-}function(x) & \to \text{Z-PREFIX-FUNCTION} \\ mk\text{-}postfix\text{-}function(x) & \to \text{Z-POSTFIX-FUNCTION} \\ mk\text{-}right\text{-}operator(x) & \to \text{Z-RIGHT-OPERATOR} \\ mk\text{-}operator(x) & \to \text{Z-OPERATOR} \\ mk\text{-}concrete\text{-}terminal(x) & \to \text{Z-CONCRETE-TERMINAL} \\ mk\text{-}basename\text{-}parts(x) & \to \text{Z-BASENAME} \end{array} \\ & & \textsf{end} \end{array}$$

The classification of a terminal symbol of class c precedes the symbol itself in the composite object produced by the *PROCESS-CONCRETE-TERMINAL* operation. Both are necessary; although $classify(c)$ could be derived from c itself it is preferable to precalculate the higher-level classification in order to avoid repeated calculation by the parser.

$$PROCESS\text{-}CONCRETE\text{-}TERMINAL\ ()$$

$$\begin{array}{l} \textsf{ext wr } input\colon Terminal\text{-}Symbol^* \\ \quad \textsf{wr } output\colon Terminal^* \\ \textsf{pre } class(\textsf{hd } input) \notin (Basename\text{-}Parts \cup \{Explan\text{-}Text\}) \\ \textsf{post let } mk\text{-}Terminal\text{-}Symbol(s, c, p) = \textsf{hd } \overleftarrow{input} \textsf{ in} \\ \qquad input = \textsf{tl } \overleftarrow{input} \wedge \\ \qquad output = \overleftarrow{output} \curvearrowright [mk\text{-}Concrete\text{-}Terminal\text{-}Symbol(classify(c), c, p)] \end{array}$$

9.1.5 Basenames

Basenames (identifiers) are more structured in Z than in many other formal languages. They are composed of a non-empty sequence of alphanumerics optionally prefixed by a Greek letter and, again optionally, followed by a subscript or superscript. The VDM-SL notation for a non-empty sequence is X^+. Basenames may also be decorated by a prime, an exclamation mark, or a query—denoting state change, output and input respectively.

$$\begin{array}{lll} \mathit{Basename} & :: & \mathit{special\text{-}char}\colon [\mathit{Basename\text{-}Part}] \\ & & \mathit{base}\colon \mathit{Name\text{-}Or\text{-}Number} \\ & & \mathit{script\text{-}decoration}\colon [\mathit{Sub\text{-}Or\text{-}Super\text{-}Script}] \\ & & \mathit{decorations}\colon \mathit{Decoration}^* \\ & & \mathit{position}\colon \mathit{Document\text{-}Position} \end{array}$$

where

$$\begin{array}{l} \mathit{inv\text{-}Basename}(\mathit{mk\text{-}Basename}(sc, b, sd, p)) \quad \triangleq \\ \qquad sc = \mathsf{nil} \lor sc \in \{pi, xi, \mathit{delta\text{-}sy}, \mathit{sigma}\} \end{array}$$

$$\mathit{Name\text{-}Or\text{-}Number} = \mathit{String}^+$$

$$\begin{array}{lll} \mathit{Sub\text{-}Or\text{-}Super\text{-}Script} & :: & \mathit{class}\colon \{\textsc{Subscript}, \textsc{Superscript}\} \\ & & \mathit{script}\colon \mathit{Name\text{-}Or\text{-}Number} \end{array}$$

$$\mathit{Decoration} = \{\mathit{Prime}, \mathit{Input}, \mathit{Output}\}$$

Consequently, the processing of basenames involves the detection and recognition of the optional components and the selection of the spelling components of the terminal symbols. The optional suffix for the basename means that the specification must request that the longest possible prefix of the input sequence is to be taken.

$$\begin{array}{l} \mathit{PROCESS\text{-}BASENAME}\ () \\ \mathsf{ext\ wr}\ \mathit{input}\colon \mathit{Terminal\text{-}Symbol}^* \\ \quad \mathsf{wr}\ \mathit{output}\colon \mathit{Terminal}^* \\ \mathsf{pre}\ \exists n \in \mathbb{N} \cdot \mathit{input}(1, \ldots, n) \in \mathit{Basename\text{-}Set} \\ \mathsf{post\ let}\ n = \mathit{max}(\{n\colon \mathbb{N} \mid \overleftarrow{\mathit{input}}(1, \ldots, n) \in \mathit{Basename\text{-}Set}\})\ \mathsf{in} \\ \qquad \mathsf{let}\ b = \overleftarrow{\mathit{input}}(1, \ldots, n)\ \mathsf{in} \\ \qquad \overleftarrow{\mathit{input}} = b \curvearrowright \mathit{input} \land \\ \qquad \mathit{output} = \overleftarrow{\mathit{output}} \curvearrowright [\mathit{convert\text{-}basename\text{-}seq}(b)] \end{array}$$

The *convert-basename-seq* function is specified implicitly below. It converts a sequence of terminal symbols representing a basename to an equivalent single terminal. It uses four auxiliary sequence conversion functions: *class-of-first*, *spellings-of*, *convert-script-seq* and *convert-decoration-seq*.

$$
\begin{array}{l}
\mathit{convert\text{-}basename\text{-}seq}\ (s\colon \mathit{Terminal\text{-}Symbol}^*)\ t\colon \mathit{Terminal} \\[4pt]
\mathsf{pre}\ s \in \mathit{Basename\text{-}Set} \\[4pt]
\mathsf{post}\ \exists s_1, s_2, s_3, s_4 \in \mathit{Terminal\text{-}Symbol}^* \cdot \\
\qquad\qquad s = s_1 \frown s_2 \frown s_3 \frown s_4 \land \\
\qquad\qquad s_1 \in (\mathit{Special\text{-}Char} \cup \{[\,]\}) \land \\
\qquad\qquad s_2 \in \mathit{Name\text{-}Or\text{-}Natural} \land \\
\qquad\qquad s_3 \in (\mathit{Script} \cup \{[\,]\}) \land \\
\qquad\qquad s_4 \in \mathit{Decoration\text{-}Seq} \land \\
\qquad\qquad t = \mathit{mk\text{-}Basename}(\mathit{class\text{-}of\text{-}first}(s_1), \mathit{spellings\text{-}of}(s_2), \\
\qquad\qquad\qquad\qquad \mathit{convert\text{-}script\text{-}seq}(s_3), \mathit{convert\text{-}decoration\text{-}seq}(s_4), \\
\qquad\qquad\qquad\qquad \mathit{position}(\mathsf{hd}\ s))
\end{array}
$$

The *class-of-first* function returns the class of the first element in its sequence argument if it is non-empty. It returns the special value nil otherwise.

$$\mathit{class\text{-}of\text{-}first} : \mathit{Terminal\text{-}Symbol}^* \to [\mathit{Terminal\text{-}Symbol\text{-}Class}]$$

$$\mathit{class\text{-}of\text{-}first}(s) \quad \triangleq \quad \mathsf{if}\ s = [\,]\ \mathsf{then\ nil}\ \mathsf{else}\ \mathit{class}(\mathsf{hd}\ s)$$

The *spellings-of* function filters out the spelling components from a sequence of terminal symbols, returning a sequence of strings.

$$\mathit{spellings\text{-}of} : \mathit{Terminal\text{-}Symbol}^* \to \mathit{String}^*$$

$$\mathit{spellings\text{-}of}(s) \quad \triangleq \quad \mathsf{if}\ s = [\,]\ \mathsf{then}\ [\,]\ \mathsf{else}\ [\mathit{spelling}(\mathsf{hd}\ s)] \frown \mathit{spellings\text{-}of}(\mathsf{tl}\ s)$$

Superscripts and subscripts on Z identifiers are enclosed in (distinct) superscript and subscript brackets. These are removed and the spellings of the intervening basename are assembled into a sequence.

$$\mathit{convert\text{-}script\text{-}seq} : \mathit{Terminal\text{-}Symbol}^* \to [\mathit{Sub\text{-}Or\text{-}Super\text{-}Script}]$$

$$
\begin{array}{l}
\mathit{convert\text{-}script\text{-}seq}(\mathit{term\text{-}seq}) \quad \triangleq \\
\quad \mathsf{if}\ \mathit{term\text{-}seq} = [\,] \\
\quad \mathsf{then\ nil} \\
\quad \mathsf{else\ let}\ \mathit{mk\text{-}Terminal\text{-}Symbol}(\mathit{spelling}, c, p) = \mathsf{hd}\ \mathit{term\text{-}seq}\ \ \mathsf{in} \\
\qquad \mathsf{let}\ s = \mathit{term\text{-}seq}(2, \ldots, \mathsf{len}\ \mathit{term\text{-}seq} - 1)\ \ \mathsf{in} \\
\qquad \mathsf{cases}\ c\ \ \mathsf{of} \\
\qquad \mathit{start\text{-}sub} \quad \to \mathit{mk\text{-}Sub\text{-}Or\text{-}Super\text{-}Script}(\textsc{Subscript}, \mathit{spellings\text{-}of}(s)) \\
\qquad \mathit{start\text{-}super} \to \mathit{mk\text{-}Sub\text{-}Or\text{-}Super\text{-}Script}(\textsc{Superscript}, \mathit{spellings\text{-}of}(s)) \\
\qquad \mathsf{end}
\end{array}
$$

Sequences of dashes, exclamation marks and question marks are converted to sequences of primes, outputs and inputs respectively.

$$convert\text{-}decoration\text{-}seq : Terminal\text{-}Symbol^* \to Decoration^*$$

$$\begin{array}{l}
convert\text{-}decoration\text{-}seq(s) \;\; \triangleq \\
\quad \mathsf{if}\ s = [\,]\ \mathsf{then}\ [\,]\ \mathsf{else\ cases\ hd}\ s\ \mathsf{of} \\
\qquad\qquad\qquad\qquad \begin{array}{lll} dashes & \to & [Prime] \curvearrowright convert\text{-}decoration\text{-}seq(\mathsf{tl}\, s) \\ exclam & \to & [Output] \curvearrowright convert\text{-}decoration\text{-}seq(\mathsf{tl}\, s) \\ query & \to & [Input] \curvearrowright convert\text{-}decoration\text{-}seq(\mathsf{tl}\, s) \\ \mathsf{end} & & \end{array}
\end{array}$$

This completes the specification of the 'pre-parser'. The reader is referred to Appendix B for the EBNF specification of the core of the parser.

9.2 Implementation

The implementation of the pre-parser was systematically developed from the above specification. Although data reifications were considered, no proof obligations were discharged even rigorously.

The task of writing a recursive descent parser is widely held to be mechanical. Parser generators are commonplace and useful software engineering tools. The experimental nature of this development precluded the use of such tools. The fairness of the experiment is maintained by not using mechanical support tools for any part of the development. This should not be thought to be a disadvantage of using formal approaches; the reader is reminded that this is not a conventional 'formal methods' development.

The implementation of the parser was produced in a pseudo-transformational style: replacing production rules of the syntax by procedures which have similar recursive structure.

9.3 Conclusions

9.3.1 The Z Basic Library

Z is an extensible language. New datatype constructors can be defined. New functions or relations can be declared as infix, prefix or postfix operators in most dialects of Z. This is not allowed in the input language of the translator: all functions must be prefix.

The familiar definitions of functions such as those which give the head and the tail of a sequence; domain and range of a relation; set union, difference and intersection etc. are not part of the Z language. Instead they are defined in a library of general purpose datatype and function definitions. This is the Z Basic Library (Spivey, 1989).

The idea of having a basic library of useful definitions seems appealing. It allows Z to be adapted to different applications by defining datatypes and standard functions which then become the primitive operations of the specifications in this application area. For example, specification authors working in image processing applications could define a version of the Z Basic Library which contained a datatype *image* and specifications of standard image processing functions. At the time of writing, the author is involved in the production of a version of the Z Basic Library for robot programming applications which extends the work reported in (Kilpatrick *et al.*, 1990).

The use of a library limits the potential for machine processing of specifications. This has become evident in the case of the Z-to-VDM-SL translator. Elegant translation from Z to VDM-SL requires using properties of the datatypes, functions and relations which are defined in the Z Basic Library. These properties are non-trivial: they cannot be deduced by the translator. In order to produce readable VDM-SL, the translator was written for the version of the Z Basic Library given in (Spivey, 1989). Thus, the use of functions such as 'head' and 'tail' in specifications which are submitted to the translator will be taken as references to the function of the same name in (Spivey, 1989).

The Z basic library is understood to be implicitly imported at the beginning of every specification. Since the Z language is not block-structured, it is not possible to redefine the basic library functions, such as 'head' and 'tail'. Thus, a specification author cannot circumvent the restriction that the basic library is fixed by re-defining the basic functions in the specification. It is of course possible to define functions called 'hd' and 'tl'—or 'car' and 'cdr'—which may be used instead of 'head' and 'tail'.

On this topic, the author does not guarantee that it will be easy to amend the translator to operate with a different version of the basic library. This is a compromise which was made to reduce the verification and implementation burden in the development.

9.3.2 The source-to-source translator

The Z tools currently in existence include programs for printing Z specifications; generating cross-references of definitions; and type-checking. In the main, these are clerical tools. J.-R. Abrial's 'B Tool' (Abrial, 1988)—described as a 'predicate calculator'—is a Z tool in the sense that it may be used as an aid in developing Z specifications although it has a more general purpose i.e. it is not Z-specific.

Although type-checking is a significant task, it does not require as much semantic manhandling of the specification as is required by translation. The *f*UZZ type-checker only requires signatures of functions to be given since this is the minimum information required to type-check a specification. Translation operates at a deeper level: a signature and a body must be given for every definition.

Consequently, it might be said that the specification source-to-source translator is among the first non-clerical Z-specific tools. This unusual position makes it possible to make observations on the problems of machine processing Z specifications. In particular, it is maintained that the Z Basic Library limits the potential for processing Z specifications by machine.

Chapter 10

A VDM Development of a Symbol Table

This chapter records a VDM development of the symbol table for the Z-to-VDM-SL source-to-source translator. The symbol table records the identifiers declared in the specification and associates a type with each.

The symbol table specification presented here has two interesting features:

1. it permits more than one type to be associated with an identifier in order to facilitate identifier 'overloading';

2. it permits the re-declaration of identifiers at different scope levels by providing facilities to create a subordinate symbol table when a new scope level is entered. This table can be destroyed on exit.

10.1 Specification

A table is modelled by a function from identifiers to sequences of type records. A non-empty sequence of tables is used to permit the creation of subordinate tables for new scope levels. It proves convenient to have the sequence of tables be non-empty to allow all the entries to be added in the same fashion. There is no requirement to treat the addition of the first element as a special case with its addition being preceded by the creation of a new table.

$$\mathit{TYPE\text{-}RECORDSa} = \mathit{TYPE\text{-}RECORD}^*$$

$$\mathit{TABLEa} = \mathit{IDENTIFIER} \xrightarrow{m} \mathit{TYPE\text{-}RECORDSa}$$

$$\mathit{SYMBOL\text{-}TABLEa} = \mathit{TABLEa}^+$$

The state of the symbol table package may be modelled by a single symbol table. It is initially empty. The empty symbol table is a sequence of exactly one empty table.

INIT ()
ext wr st: $\mathit{SYMBOL\text{-}TABLEa}$
post $st = [\{\,\}]$

The operations to create a new table and to destroy an existing table are now defined. The *OPEN-SCOPE* operation prefixes the existing table with an empty table. The most recently added table may be removed using the *CLOSE-SCOPE* operation.

$$\begin{array}{l} \mathit{OPEN\text{-}SCOPE}\ () \\ \mathsf{ext\ wr}\ st\colon \mathit{SYMBOL\text{-}TABLEa} \\ \mathsf{post}\ st = [\{\,\}] \curvearrowright \overleftarrow{st} \end{array}$$

$$\begin{array}{l} \mathit{CLOSE\text{-}SCOPE}\ () \\ \mathsf{ext\ wr}\ st\colon \mathit{SYMBOL\text{-}TABLEa} \\ \mathsf{pre\ len}\ st > 1 \\ \mathsf{post}\ st = \mathsf{tl}\ \overleftarrow{st} \end{array}$$

The operation to add an identifier to the table is now defined. The cases where an identifier is or is not present are treated separately. If the identifier is not present, the table is extended to include the new value. If the identifier is present, its image under the function is extended to include the type from the new declaration.

Note that this operation modifies the most recently created table. This table contains declarations at the deepest level of scope.

$$\begin{array}{l} \mathit{ADD}\ (id\colon \mathit{IDENTIFIER};\ tr\colon \mathit{TYPE\text{-}RECORD}) \\ \mathsf{ext\ wr}\ st\colon \mathit{SYMBOL\text{-}TABLEa} \\ \mathsf{post}\ id \notin \mathsf{dom\ hd}\ \overleftarrow{st} \Rightarrow st = [\mathsf{hd}\ \overleftarrow{st} \cup \{id \mapsto [tr]\}] \curvearrowright \mathsf{tl}\ \overleftarrow{st} \wedge \\ \qquad id \in \mathsf{dom\ hd}\ \overleftarrow{st} \Rightarrow st = [\mathsf{hd}\ \overleftarrow{st} \dagger \{id \mapsto ([tr] \curvearrowright \mathsf{hd}\ \overleftarrow{st}(id))\}] \curvearrowright \mathsf{tl}\ \overleftarrow{st} \end{array}$$

It is necessary to be able to check if an identifier has already been declared at any level of scope. The *is-in* function enables such a search to be performed. The **inds** function is a specialization of the **dom** function for application to sequences.

$$\begin{array}{l} \mathit{is\text{-}in} : \mathit{IDENTIFIER} \times \mathit{SYMBOL\text{-}TABLEa} \to \mathbb{B} \\ \mathit{is\text{-}in}(id, st) \quad \triangleq \quad \exists i \in \mathsf{inds}\, st \cdot id \in \mathsf{dom}\, st(i) \end{array}$$

It is also necessary to be able to ascertain all possible interpretations of an identifier—there may be none. The *get-types* function enables the types to be obtained from the table.

$$\begin{array}{l} \mathit{get\text{-}types} : \mathit{IDENTIFIER} \times \mathit{SYMBOL\text{-}TABLEa} \to \mathit{TYPE\text{-}RECORDSa} \\ \mathit{get\text{-}types}(id, st) \quad \triangleq \quad \begin{array}[t]{l} \mathsf{if\ len}\ st = 1 \\ \mathsf{then\ if}\ id \in \mathsf{dom\ hd}\ st \\ \qquad \mathsf{then\ hd}\ st(id) \\ \qquad \mathsf{else}\ [\,] \\ \mathsf{else\ if}\ id \in \mathsf{dom\ hd}\ st \\ \qquad \mathsf{then\ hd}\ st(id) \curvearrowright \mathit{get\text{-}types}(id, \mathsf{tl}\ st) \\ \qquad \mathsf{else}\ \mathit{get\text{-}types}(id, \mathsf{tl}\ st) \end{array} \end{array}$$

This completes the specification of the symbol table.

10.2 Proofs about the specification

10.2.1 Proofs of implementability

The Vienna Development Method requires that the above operations be shown to be *implementable.* That is, there must be at least one possible result for all allowable inputs.

Implementability of the *INIT* operation

The proof of this result reduces to showing that the expression $[\{\}]$ is of the correct type. This proof is very simple but is useful for illustrating the natural deduction proof style favoured by VDM. This form of proof has been familiar to the program verification community since Gries' seminal work (Gries, 1981). A natural deduction proof consists of hypotheses introduced by the keyword from; a sequence of numbered deductions—which may themselves be natural deduction proofs—justified by reference to a previous deduction or a property of the operator, function or operation used; and a concluding inference proclaimed by the keyword infer. The subproofs may be marked by indentation to amplify the structure of the proof. Here, the subproofs are indented; numbers appear on the left; and justifications appear on the right.

	from $\{\} \in IDENTIFIER \xrightarrow{m} TYPE\text{-}RECORDSa$	
1	len $[\{\}] = 1$	property of len
	infer $[\{\}] \in SYMBOL\text{-}TABLEa$	hyp, 1

Implementability of the *OPEN-SCOPE* operation

The implementability lemma to be proved here is:

$$\forall \overleftarrow{st} \in SYMBOL\text{-}TABLEa \cdot [\{\}] \frown \overleftarrow{st} \in SYMBOL\text{-}TABLEa$$

The proof of this result simply relates in a natural deduction style the argument that the concatenation of two non-empty sequences is a non-empty sequence. Because the concatenation operator may only be applied to sequences of identical type, the proof is complete.

	from $[\{\}] \in SYMBOL\text{-}TABLEa, \overleftarrow{st} \in SYMBOL\text{-}TABLEa$	
1	len $[\{\}] \geqslant 1 \wedge$ len $\overleftarrow{st} \geqslant 1$	hyp
2	len $[\{\}] \frown \overleftarrow{st} > 1$	1
	infer $[\{\}] \frown \overleftarrow{st} \in SYMBOL\text{-}TABLEa$	

Implementability of the *CLOSE-SCOPE* operation

The implementability lemma to be proved here is:

$$\forall \overleftarrow{st} \in \mathit{SYMBOL\text{-}TABLEa} \cdot \mathsf{len}\ \overleftarrow{st} > 1 \ \Rightarrow\ \mathsf{tl}\ \overleftarrow{st} \in \mathit{SYMBOL\text{-}TABLEa}$$

The *CLOSE-SCOPE* operation has an execution pre-condition. The operation may only be invoked if the symbol table st contains at least two elements. This pre-condition is taken as a hypothesis in the proof. One small change must be made to the statement of the pre-condition since in VDM-SL, state-before variables are hooked in the post-condition but not in the pre-condition. To maintain consistency of reference throughout the proof, the pre-condition variable is also hooked.

	from $\overleftarrow{st} \in \mathit{SYMBOL\text{-}TABLEa}, \mathsf{len}\ \overleftarrow{st} > 1$	
1	$\quad \mathsf{len}\ \mathsf{tl}\ \overleftarrow{st} \geqslant 1$	hyp, prop of tl
	infer $\mathsf{tl}\ \overleftarrow{st} \in \mathit{SYMBOL\text{-}TABLEa}$	

Implementability of the *ADD* operation

The implementability lemma for this operation is:

$$\begin{array}{l} \forall id \in \mathit{IDENTIFIER}, tr \in \mathit{type\text{-}record}, \overleftarrow{st} \in \mathit{SYMBOL\text{-}TABLEa} \cdot \\ \quad \exists st \in \mathit{SYMBOL\text{-}TABLEa} \cdot \\ \quad\quad \mathit{post\text{-}ADD}(id, tr, \overleftarrow{st}, st) \end{array}$$

The predicate *post-ADD* above denotes the post-condition of the *ADD* operation. The post-condition of this operation is considerably more complex than the post-conditions of the previous operations. Consequently, the proof of implementability is more complex than the previous implementability proofs.

The structure of the post-condition is reflected in the structure of the proof. The post-condition consists of two conjoined implications and the proof consists of two subproofs with the corresponding antecedent as the hypothesis of the subproof. The proof is given in Figure 10.1.

Implementability of the *is-in* function

The implementability lemma for this function requires the result of the function to be of the correct type for all possible function arguments.

$$\begin{array}{l} \forall id \in \mathit{IDENTIFIER}, st \in \mathit{SYMBOL\text{-}TABLEa} \cdot \\ \quad (\exists i \in \mathsf{inds}\ st \cdot id \in \mathsf{dom}\ st(i)) \in \mathbb{B} \end{array}$$

The proof of this result constructs the existentially quantified expression in steps which determine the type of the subexpressions.

from $id \in IDENTIFIER, tr \in type\text{-}record, \overleftarrow{st} \in SYMBOL\text{-}TABLEa$

1 from $id \notin \mathsf{dom}\ \mathsf{hd}\ \overleftarrow{st}$

1.1 $\mathsf{hd}\ \overleftarrow{st} \cup \{id \mapsto [tr]\} \in TABLEa$ hyp(1)

1.2 $[\mathsf{hd}\ \overleftarrow{st} \cup \{id \mapsto [tr]\}] \in SYMBOL\text{-}TABLEa$ property of [], 1.1

1.3 $\mathsf{tl}\ \overleftarrow{st} \in TABLEa^*$ hyps

1.4 $[\mathsf{hd}\ \overleftarrow{st} \cup \{id \mapsto [tr]\}] \curvearrowright \mathsf{tl}\ \overleftarrow{st} \in SYMBOL\text{-}TABLEa$ $\curvearrowright$-defn, 1.2, 1.3

infer $\exists st \in SYMBOL\text{-}TABLEa \cdot st = [\mathsf{hd}\ \overleftarrow{st} \cup \{id \mapsto [tr]\}] \curvearrowright \mathsf{tl}\ \overleftarrow{st}$

2 from $id \in \mathsf{dom}\ \mathsf{hd}\ \overleftarrow{st}$

2.1 $\mathsf{hd}\ \overleftarrow{st} \dagger \{id \mapsto ([tr] \curvearrowright \mathsf{hd}\ \overleftarrow{st}(id))\} \in TABLEa$ hyp(2)

2.2 $[\mathsf{hd}\ \overleftarrow{st} \dagger \{id \mapsto ([tr] \curvearrowright \mathsf{hd}\ \overleftarrow{st}(id))\}] \in SYMBOL\text{-}TABLEa$
property of [], 2.1

2.3 $\mathsf{tl}\ \overleftarrow{st} \in TABLEa^*$ hyps

2.4 $[\mathsf{hd}\ \overleftarrow{st} \dagger \{id \mapsto ([tr] \curvearrowright \mathsf{hd}\ \overleftarrow{st}(id))\}] \curvearrowright \mathsf{tl}\ \overleftarrow{st} \in SYMBOL\text{-}TABLEa$
$\curvearrowright$-defn, 2.2, 2.3

infer $\exists\, st \in SYMBOL\text{-}TABLEa \cdot$
$st = [\mathsf{hd}\ \overleftarrow{st} \dagger \{id \mapsto ([tr] \curvearrowright \mathsf{hd}\ \overleftarrow{st}(id))\}] \curvearrowright \mathsf{tl}\ \overleftarrow{st}$

infer $\exists st \in SYMBOL\text{-}TABLEa \cdot post\text{-}ADD(id, tr, \overleftarrow{st}, st)$

Figure 10.1: Implementability of *ADD*.

	from $id \in \mathit{IDENTIFIER}, st \in \mathit{SYMBOL\text{-}TABLEa}, i \in \mathsf{inds}\, st$	
1	$\quad st(i) \in \mathit{TABLEa}$	hyp, $\mathit{SYMBOL\text{-}TABLEa}$-defn
2	$\quad \mathsf{dom}\, st(i) \in \mathit{IDENTIFIER}\text{-}\mathsf{set}$	1, TABLEa-defn
3	$\quad (id \in \mathsf{dom}\, st(i)) \in \mathbb{B}$	hyp, 2, property of $\in$
	infer $(\exists i \in \mathsf{inds}\, st \cdot id \in \mathsf{dom}\, st(i)) \in \mathbb{B}$	

Implementability of the *get-types* function

The implementability lemma for this function requires the result of the function to be of the correct type for all possible function arguments.

$$\forall id \in \mathit{IDENTIFIER}, st \in \mathit{SYMBOL\text{-}TABLEa} \cdot \\ \quad \mathit{get\text{-}types}(id, st) \in \mathit{TYPE\text{-}RECORDSa}$$

This proof is slightly more substantial than the previous proofs and is conducted by induction over the length of a *SYMBOL-TABLEa* argument. As before, the structure of the proof echoes the structure of the function definition. A base case ($\mathsf{len}\, st = 1$) and an induction step ($\mathsf{len}\, st > 1$) are identified. Within these, the proof proceeds by case analysis. The first case is $id \in \mathsf{dom}\,\mathsf{hd}\, st$, the second is $id \notin \mathsf{dom}\,\mathsf{hd}\, st$. The proof is given in Figure 10.2.

10.2.2 Proofs of properties

The Vienna Development Method requires the above implementability proofs to be constructed. However, a disciplined specification author may wish to prove further properties of the specification which are unique to the present specification and could not be predicted by a development method. One additional simple proof will be constructed.

The following proof establishes the result that the *ADD* operation only modifies the most recent created table.

	from $\mathit{post\text{-}ADD}(\overleftarrow{st}, id, tr, st)$	
1	$\quad$ from $id \notin \mathsf{dom}\,\mathsf{hd}\, \overleftarrow{st}$	
1.1	$\qquad st = [\mathsf{hd}\, \overleftarrow{st} \cup \{id \mapsto [tr]\}] \frown \mathsf{tl}\, \overleftarrow{st}$	$\mathit{post\text{-}ADD}$
	$\quad$ infer $\mathsf{tl}\, st = \mathsf{tl}\, \overleftarrow{st}$	
2	$\quad$ from $id \in \mathsf{dom}\,\mathsf{hd}\, \overleftarrow{st}$	
2.1	$\qquad st = [\mathsf{hd}\, \overleftarrow{st} \dagger \{id \mapsto ([tr] \frown \mathsf{hd}\, \overleftarrow{st}(id))\}] \frown \mathsf{tl}\, \overleftarrow{st}$	$\mathit{post\text{-}ADD}$
	$\quad$ infer $\mathsf{tl}\, st = \mathsf{tl}\, \overleftarrow{st}$	
	infer $\mathsf{tl}\, st = \mathsf{tl}\, \overleftarrow{st}$	

From this, conclude that $\mathit{post\text{-}ADD}(\overleftarrow{st}, id, tr, st) \vdash \mathsf{tl}\, st = \mathsf{tl}\, \overleftarrow{st}$. Hence, the *ADD* operation changes at most the head of the sequence. This is the most recently created table.

$$
\begin{array}{lll}
 & \textsf{from } id \in \mathit{IDENTIFIER}, st \in \mathit{SYMBOL\text{-}TABLEa}, & \\
1 & \quad \textsf{from } \textsf{len}\, st = 1 & \\
1.1 & \qquad \textsf{from } id \in \textsf{dom}\,\textsf{hd}\, st & \\
1.1.1 & \qquad\quad \textsf{hd}\, st(id) \in \mathit{TYPE\text{-}RECORDSa} & \text{hyp}(1.1) \\
 & \qquad \textsf{infer } \mathit{get\text{-}types}(id, st) \in \mathit{TYPE\text{-}RECORDSa} & \\
1.2 & \qquad \textsf{from } id \notin \textsf{dom}\,\textsf{hd}\, st & \\
1.2.1 & \qquad\quad [\,] \in \mathit{TYPE\text{-}RECORDSa} & \text{property of sequences} \\
 & \qquad \textsf{infer } \mathit{get\text{-}types}(id, st) \in \mathit{TYPE\text{-}RECORDSa} & \\
 & \quad \textsf{infer } \mathit{get\text{-}types}(id, st) \in \mathit{TYPE\text{-}RECORDSa} & \\
2 & \quad \textsf{from } \textsf{len}\, st > 1, \mathit{get\text{-}types}(id, \textsf{tl}\, st) \in \mathit{TYPE\text{-}RECORDSa} & \\
2.1 & \qquad \textsf{from } id \in \textsf{dom}\,\textsf{hd}\, st & \\
2.1.1 & \qquad\quad \textsf{hd}\, st(id) \in \mathit{TYPE\text{-}RECORDSa} & \text{hyp}(2.1) \\
2.1.2 & \qquad\quad \mathit{get\text{-}types}(id, \textsf{tl}\, st) \in \mathit{TYPE\text{-}RECORDSa} & \text{hyp}(2) \\
2.1.3 & \qquad\quad \textsf{hd}\, st(id) \curvearrowright \mathit{get\text{-}types}(id, \textsf{tl}\, st) \in \mathit{TYPE\text{-}RECORDSa} & \\
 & & \text{property of } \curvearrowright, 2.1.1, 2.1.2 \\
 & \qquad \textsf{infer } \mathit{get\text{-}types}(id, st) \in \mathit{TYPE\text{-}RECORDSa} & \\
2.2 & \qquad \textsf{from } id \notin \textsf{dom}\,\textsf{hd}\, st & \\
2.2.1 & \qquad\quad \mathit{get\text{-}types}(id, \textsf{tl}\, st) \in \mathit{TYPE\text{-}RECORDSa} & \text{hyp}(2) \\
 & \qquad \textsf{infer } \mathit{get\text{-}types}(id, st) \in \mathit{TYPE\text{-}RECORDSa} & \\
 & \quad \textsf{infer } \mathit{get\text{-}types}(id, st) \in \mathit{TYPE\text{-}RECORDSa} & \\
 & \textsf{infer } \forall\, id \in \mathit{IDENTIFIER}, st \in \mathit{SYMBOL\text{-}TABLEa} \cdot & \\
 & \qquad\qquad \mathit{get\text{-}types}(id, st) \in \mathit{TYPE\text{-}RECORDSa} &
\end{array}
$$

Figure 10.2: Implementability of *get-types*.

10.3 Data reification

The major reification problem posed by the above specification is the use of maps in the model of tables and symbol tables. Conventional programming languages do not provide maps as a primitive data type. Data reification is used to replace the maps by ordered binary trees. Binary search trees may be easily described using conventional programming languages.

$$TYPE\text{-}RECORDSb = TYPE\text{-}RECORD^*$$

$$\begin{array}{ll} NODEb :: & key: IDENTIFIER, \\ & value: TYPE\text{-}RECORDSb, \\ & left, right: TABLEb \end{array}$$

$$TABLEb = [NODEb]$$

where

$$\begin{array}{l} inv\text{-}TABLEb(t) \quad \triangleq \quad t \neq \mathsf{nil} \Rightarrow \\ \quad (\forall lk \in keys(left(t)) \cdot lk < key(t) \wedge \\ \quad \forall rk \in keys(right(t)) \cdot key(t) < rk) \end{array}$$

$$keys : TABLEb \rightarrow IDENTIFIER\text{-}\mathsf{set}$$

$$keys(t) \quad \triangleq \quad \mathsf{if}\ t = \mathsf{nil}\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ keys(left(t)) \cup \{key(t)\} \cup keys(right(t))$$

$$SYMBOL\text{-}TABLEb = TABLEb^+$$

Having produced these new models for tables and symbol tables, it is now necessary to state the relationship between these and the earlier models. The method used by VDM to record such relationships is to provide a *retrieve function* to document how a value of the concrete type is mapped to the corresponding value in the abstract type. The following function, *retr-TABLEa* is the retrieve function for the *TABLEa* datatype.

$$retr\text{-}TABLEa : TABLEb \rightarrow TABLEa$$

$$\begin{array}{l} retr\text{-}TABLEa(t) \quad \triangleq \\ \quad \mathsf{if}\ t = \mathsf{nil}\ \mathsf{then}\ \{\,\} \\ \quad \mathsf{else}\ retr\text{-}TABLEa(left(t)) \dagger \{key(t) \mapsto value(t)\} \dagger retr\text{-}TABLEa(right(t)) \end{array}$$

Because the *SYMBOL-TABLE* types use the *TABLE* types, it is convenient to state the retrieve function for the *SYMBOL-TABLE* type in terms of the *retr-TABLEa* function.

$$retr\text{-}SYMBOL\text{-}TABLEa : SYMBOL\text{-}TABLEb \rightarrow SYMBOL\text{-}TABLEa$$

$$\begin{array}{l} retr\text{-}SYMBOL\text{-}TABLEa(st) \quad \triangleq \\ \quad \mathsf{if}\ \mathsf{len}\ st = 1\ \mathsf{then}\ [retr\text{-}TABLEa(\mathsf{hd}\ st)] \\ \quad \mathsf{else}\ [retr\text{-}TABLEa(\mathsf{hd}\ st)] \curvearrowright retr\text{-}SYMBOL\text{-}TABLEa(\mathsf{tl}\ st) \end{array}$$

It is now necessary to rewrite the operations of the specification appropriately for action upon the new models of tables and symbol tables.

$INIT\ ()$
$\mathsf{ext\ wr}\ st\colon SYMBOL\text{-}TABLEb$
$\mathsf{post}\ st = [\mathsf{nil}]$

$OPEN\text{-}SCOPE\ ()$
$\mathsf{ext\ wr}\ st\colon SYMBOL\text{-}TABLEb$
$\mathsf{post}\ st = [\mathsf{nil}] \frown \overleftarrow{st}$

$CLOSE\text{-}SCOPE\ ()$
$\mathsf{ext\ wr}\ st\colon SYMBOL\text{-}TABLEb$
$\mathsf{pre\ len}\ st > 1$
$\mathsf{post}\ st = \mathsf{tl}\ \overleftarrow{st}$

The *ADD* operation now becomes much more complex. A subordinate *insert* function is introduced to structure the statement of the operation.

$ADD\ (id\colon IDENTIFIER;\ tr\colon TYPE\text{-}RECORD)$
$\mathsf{ext\ wr}\ st\colon SYMBOL\text{-}TABLEb$
$\mathsf{post}\ st = [insert(\mathsf{hd}\ \overleftarrow{st}, id, tr)] \frown \mathsf{tl}\ \overleftarrow{st}$

$insert : TABLEb \times IDENTIFIER \times TYPE\text{-}RECORD \to TABLEb$

$$\begin{array}{l} insert(t, id, tr) \quad \triangleq \\ \quad \mathsf{if}\ t = \mathsf{nil} \\ \quad \mathsf{then}\ mk\text{-}TABLEb(id, [tr], \mathsf{nil}, \mathsf{nil}) \\ \quad \mathsf{else\ if}\ id < key(t) \\ \qquad \mathsf{then}\ mk\text{-}TABLEb(key(t), value(t), insert(left(t), id, tr), right(t)) \\ \qquad \mathsf{else\ if}\ key(t) < id \\ \qquad\quad \mathsf{then}\ mk\text{-}TABLEb(key(t), value(t), left(t), insert(right(t), id, tr)) \\ \qquad\quad \mathsf{else}\ mk\text{-}TABLEb(key(t), [tr] \frown value(t), left(t), right(t)) \end{array}$$

It is desirable to remove the existential quantifier from the statement of the *is-in* function. Conventional programming languages do not provide existential quantifiers. It is possible to take the simplistic approach of replacing the quantifier by its meaning—generalised disjunction—because the scope of the quantification is small. The scope of quantification represents the number of levels of scope used in the text being processed. This would rarely be more than 10. An *in-table* function is introduced because the dom function may no longer be applied to the symbol table.

$is\text{-}in : IDENTIFIER \times SYMBOL\text{-}TABLEb \to \mathbb{B}$

$$\begin{array}{ll} is\text{-}in(id, st) \quad \triangleq & \mathsf{if\ len}\ st = 1 \\ & \mathsf{then}\ in\text{-}table(id, \mathsf{hd}\ st) \\ & \mathsf{else}\ in\text{-}table(id, \mathsf{hd}\ st) \lor is\text{-}in(id, \mathsf{tl}\ st) \end{array}$$

$$in\text{-}table : IDENTIFIER \times TABLEb \rightarrow \mathbb{B}$$

$$\begin{array}{lll} in\text{-}table(id, t) & \triangleq & \textsf{if } t = \textsf{nil} \\ & & \textsf{then false} \\ & & \textsf{else if } id < key(t) \\ & & \quad \textsf{then } in\text{-}table(id, left(t)) \\ & & \quad \textsf{else if } key(t) < id \\ & & \quad\quad \textsf{then } in\text{-}table(id, right(t)) \\ & & \quad\quad \textsf{else true} \end{array}$$

Replacing the maps by binary trees means that map application cannot be employed in the *get-types* function. Instead, an algorithm must be devised to search for the types which are associated with an identifier. A *types-of* function is introduced for this purpose.

$$get\text{-}types : IDENTIFIER \times SYMBOL\text{-}TABLEb \rightarrow TYPE\text{-}RECORDSb$$

$$\begin{array}{lll} get\text{-}types(id, st) & \triangleq & \textsf{if len } st = 1 \\ & & \textsf{then } types\text{-}of(id, \textsf{hd } st) \\ & & \textsf{else } types\text{-}of(id, \textsf{hd } st) \curvearrowright get\text{-}types(id, \textsf{tl } st) \end{array}$$

The *types-of* function is a total function which returns the empty sequence if the identifier cannot be found.

$$types\text{-}of : IDENTIFIER \times TABLEb \rightarrow TYPE\text{-}RECORDSb$$

$$\begin{array}{lll} types\text{-}of(id, t) & \triangleq & \textsf{if } t = \textsf{nil} \\ & & \textsf{then } [\,] \\ & & \textsf{else if } id < key(t) \\ & & \quad \textsf{then } types\text{-}of(id, left(t)) \\ & & \quad \textsf{else if } key(t) < id \\ & & \quad\quad \textsf{then } types\text{-}of(id, right(t)) \\ & & \quad\quad \textsf{else } value(t) \end{array}$$

It is now necessary to prove that the meaning of the specification has not been changed by this data reification.

10.4 Proofs about this design step

10.4.1 Adequacy of the retrieve functions

The retrieve functions given above must be shown to be total surjective functions.

The *retr-TABLEa* function may be easily shown to be total. The proof that this function is surjective is by map induction and is given in Figure 10.3.

The *retr-SYMBOL-TABLEa* function merely applies the *retr-TABLEa* function to every element of its sequence argument and is taken to be adequate. Adequacy proofs would be conducted by induction over the length of a *SYMBOL-TABLEa* element and would rely on the adequacy of the *retr-TABLEa* function.

from $t_1 \in \mathit{TABLEa}$

1 **from** $t_1 = \{\,\}$

1.1 $\mathit{retr\text{-}TABLEa}(\mathsf{nil}) = \{\,\}$ definition of *retr-TABLEa*

infer $\exists\, t_2 \in \mathit{TABLEb} \cdot \mathit{retr\text{-}TABLEa}(t_2) = t_1$

2 **from** $t_1 = m_1 \dagger \{k \mapsto v\} \dagger m_2,$

$\exists\, t_3 \in \mathit{TABLEb} \cdot \mathit{retr\text{-}TABLEa}(t_3) = m_1,$

$\exists\, t_4 \in \mathit{TABLEb} \cdot \mathit{retr\text{-}TABLEa}(t_4) = m_2,$

$\forall\, lk \in \mathsf{dom}\, m_1 \cdot lk < k,$

$\forall\, rk \in \mathsf{dom}\, m_2 \cdot k < rk$

2.1 $\forall\, m \in \mathit{TABLEa}, t \in \mathit{TABLEb} \cdot$

$\mathit{retr\text{-}TABLEa}(t) = m \Rightarrow \mathit{keys}(t) = \mathsf{dom}\, m$ property of *keys*

2.2 $\exists\, t_3 \in \mathit{TABLEb} \cdot$

$\mathit{retr\text{-}TABLEa}(t_3) = m_1 \wedge \forall lk \in \mathit{keys}(t_3) \cdot lk < k$ hyp(2), 2.1

2.3 $\exists\, t_4 \in \mathit{TABLEb} \cdot$

$\mathit{retr\text{-}TABLEa}(t_4) = m_2 \wedge \forall rk \in \mathit{keys}(t_4) \cdot k < rk$ hyp(2), 2.1

2.4 $\exists\, t_3, t_4 \in \mathit{TABLEb} \cdot$

$\mathit{retr\text{-}TABLEa}(\mathit{mk\text{-}NODEb}(k, v, t_3, t_4)) = m_1 \dagger \{k \mapsto v\} \dagger m_2$

2.2, 2.3

infer $\exists\, t_2 \in \mathit{TABLEb} \cdot \mathit{retr\text{-}TABLEa}(t_2) = t_1$

infer $\forall\, t_1 \in \mathit{TABLEa} \cdot \exists\, t_2 \in \mathit{TABLEb} \cdot \mathit{retr\text{-}TABLEa}(t_2) = t_1$

Figure 10.3: Adequacy of *retr-TABLEa*.

10.4.2 Implementability and satisfaction lemmas

In this section the proofs ensure that the data reification has not altered the meaning of the operations and functions of the specification.

Proofs of the implementability of the operations and functions are again required because it is possible to produce adequate refinements of operations and functions which are not implementable. The proofs are straightforward and are omitted.

The *INIT*, *OPEN-SCOPE* and *CLOSE-SCOPE* operations

The proof that the *INIT* and *OPEN-SCOPE* operations are consistent with the earlier ones depends on the assumption that the *retr-TABLEa* function maps nil to the empty map. This is obvious from the definition of *retr-TABLEa*.

The proof of the reification of *CLOSE-SCOPE* is produced in two steps. Firstly, the domain rule for operations is used to show that the new operation produces a well-defined result at least every time that the previous version produced a well-defined result. Secondly, the result rule is used to show that every time that the new operation succeeds it produces an acceptable answer.

Domain Lemma:

The domain rule states that the pre-condition of the new operation should not be stronger than the pre-condition of the old operation. For the *CLOSE-SCOPE* operation, this produces the following proof obligation.

$$\begin{array}{l} \vdash \; \forall st \in \mathit{SYMBOL\text{-}TABLEb} \cdot \\ \quad \mathsf{len}\; \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(st) > 1 \Rightarrow \mathsf{len}\; st > 1 \end{array}$$

Proof of this result may be furnished by noting that the *retr-SYMBOL-TABLEa* function returns a sequence of the same length as its argument.

Result Lemma:

The result rule for operations requires proof that any result produced by the reified operation is acceptable to the old operation. The following property of the *CLOSE-SCOPE* operation is established.

$$\begin{array}{l} \vdash \; \forall \overleftarrow{st}, st \in \mathit{SYMBOL\text{-}TABLEb} \cdot \\ \quad \mathsf{len}\; \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(\overleftarrow{st}) > 1 \wedge st = \mathsf{tl}\; \overleftarrow{st} \Rightarrow \\ \qquad \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(st) = \mathsf{tl}\; \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(\overleftarrow{st}) \end{array}$$

The proof of this property appeals to the fact that the *retr-SYMBOL-TABLEa* function does not conflict with the tl function: the two functions may be applied in any order.

$$
\begin{array}{lll}
 & \textsf{from } \overleftarrow{st}, st \in \mathit{SYMBOL\text{-}TABLEb}, & \\
 & \quad \textsf{len } \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(\overleftarrow{st}) > 1, & \\
 & \quad st = \textsf{tl } \overleftarrow{st} & \\
1 & \quad \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(st) = \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(\textsf{tl } \overleftarrow{st}) & \text{hyp(1)} \\
 & \textsf{infer } \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(st) = \textsf{tl } \mathit{retr\text{-}SYMBOL\text{-}TABLEa}(\overleftarrow{st}) & \\
 & & \text{property of } \mathit{retr\text{-}SYMBOL\text{-}TABLEa}
\end{array}
$$

Satisfaction proof for the *ADD* operation

The change to the *ADD* operation is sufficiently large to make the proof of satisfaction long. Only an outline of the proof is presented here.

1. When given a table which does not contain *id*, the *insert* function will create a sub-tree of the form $\mathit{mk\text{-}TABLEb}(id, [tr], \textsf{nil}, \textsf{nil})$. This corresponds to $\{id \mapsto [tr]\}$ under *retr-TABLEa*.

2. When given a table containing *id*, the value at the relevant node is updated to $[tr] \curvearrowright$ "old value". This corresponds to the $\{id \mapsto [tr] \curvearrowright$ "old value"$\}$ case in the previous formulation of *ADD*.

10.4.3 Function satisfaction for *is-in* and *get-types*

The implementability of the functions of the specification ensures that they are total. It remains to show that the versions for binary trees produce answers which are acceptable to the versions for maps.

The *is-in* function will obviously distribute the search for an identifier in a symbol table throughout the sequence of tables which make up the symbol table. The core of the proof of satisfaction for the *is-in* function lies in establishing that the *in-table* function will find an identifier *id* in a table *t* if it is present:

$$\vdash \; \mathit{in\text{-}table}(id, t) \Leftrightarrow id \in \textsf{dom } \mathit{retr\text{-}TABLEa}(t)$$

Consider a simpler version of *in-table*.

$$\mathit{in\text{-}table} : \mathit{IDENTIFIER} \times \mathit{TABLEb} \to \mathbb{B}$$

$$
\begin{array}{lll}
\mathit{in\text{-}table}(id, t) & \triangleq & \textsf{if } t = \textsf{nil} \textsf{ then false} \\
 & & \textsf{else } \mathit{in\text{-}table}(id, \mathit{left}(t)) \vee \mathit{key}(t) = id \vee \mathit{in\text{-}table}(id, \mathit{right}(t))
\end{array}
$$

This simplified version will obviously find the identifier *id* in *t* if it is present. The earlier version will achieve the same result by using the *TABLEb* data type invariant to decide which sub-tree to search for *id*. Consequently, it will also find the identifier *id* in *t* if it is present.

The *get-types* function has a similar structure to *is-in* and a proof of its correctness would be similar to the proof given for *is-in*. The proof is omitted.

This completes the proofs of the lemmas relating to this reification of the specification.

10.5 Language reification

The above specification contains enough computational information for it to be considered a program. It uses only simple data types and direct definitions are given for all functions. In addition, the post-conditions of the operations are sufficiently simple to suggest obvious implementations.

The data reification performed was sufficient to cast the functions and operations in the executable subset of the VDM-SL specification language. It now only remains to re-state the specification in a programming language. Such a step may be termed a *language reification* because the majority of the work is concerned with refining the functions and operations rather than the data types.

This final step—although simpler than the previous one—is still error prone. It is necessary to continue to consider the correctness of the result and furnish proof that the Ada functions and procedures satisfy the VDM-SL functions and operations of the previous refinement of the specification.

10.5.1 Data types

Ada equivalents are now given for the VDM-SL data types used in the most recent reification of the specification. Sequences of type records are modelled by the obvious linked lists.

```
type Type_Records is access Type_Record_Element;
type Type_Record_Element is
   record
      hd: Type_Record;
      tl: Type_Records;
   end record;
type Type_Record is ...;
```

The binary tree table type is modelled by the obvious Ada record containing pointers to left and right sub-trees. The data type invariant is included as a comment. It appeals to the invariant function for VDM-SL trees using a retrieve function for tables defined later.

```
type Node is
   record
      key: Identifier;
      value: Type_Records;
      left, right: Table;
   end record;
type Identifier is ...;
type Table is access Node;
-- where inv-Table(t) △ inv-TABLEb(retr-TABLEb(t))
```

The symbol table type is modelled by a (possibly empty) linked list of records which have a head component which is a table and a symbol table tail component. These represent the results returned by hd and tl respectively. Because this representation does not have a data type invariant, the pre-conditions of operations and functions must be strengthened to account for the *null* value.

type *Symbol_Table* is access *Symbol_Table_Element*;
type *Symbol_Table_Element* is
  record
    hd: *Table*;
    tl: *Symbol_Table*;
  end record;

The *st* variable is simply a variable of type *Symbol_Table*.

st: *Symbol_Table*;

VDM retrieve functions are now used to relate the Ada data types to the VDM data types. The retrieve function for type records simply moves through the linked list making up a sequence of the type records it finds.

$$retr\text{-}TYPE\text{-}RECORDSb : Type\_Records \to TYPE\text{-}RECORDSb$$

$$retr\text{-}TYPE\text{-}RECORDSb(tr) \;\triangleq\; \textsf{if } tr = nil \textsf{ then } [\,] \textsf{ else } [tr.hd] \curvearrowright retr\text{-}TYPE\text{-}RECORDSb(tr.tl)$$

The *retr-TABLEb* function traverses the tree converting values by applying the retrieve function for the type *TYPE-RECORDS* and converting sub-trees recursively. As the base case for the recursion, the Ada nil pointer value *null* is converted to the VDM-SL nil constant nil.

$$retr\text{-}TABLEb : Table \to TABLEb$$

$$\begin{aligned}
&retr\text{-}TABLEb(t) \;\triangleq\\
&\quad \textsf{if } t = null\\
&\quad \textsf{then nil}\\
&\quad \textsf{else } mk\text{-}NODEb(t.key, retr\text{-}TYPE\text{-}RECORDSb(t.value),\\
&\qquad\quad retr\text{-}TABLEb(t.left), retr\text{-}TABLEb(t.right))
\end{aligned}$$

Symbol tables are only slightly more difficult to convert than sequences of type records. Totality must be ensured by defining an image for *null*. This value must be a non-empty sequence in order to satisfy the data type invariant. Choose [nil].

$$retr\text{-}SYMBOL\text{-}TABLEb : Symbol\_Table \to SYMBOL\text{-}TABLEb$$

$$\begin{aligned}
&retr\text{-}SYMBOL\text{-}TABLEb(st) \;\triangleq\\
&\quad \textsf{if } st = null\\
&\quad \textsf{then [nil]}\\
&\quad \textsf{else if } st.tl = null\\
&\qquad \textsf{then } [retr\text{-}TABLEb(st.hd)]\\
&\qquad \textsf{else } [retr\text{-}TABLEb(st.hd)] \curvearrowright retr\text{-}SYMBOL\text{-}TABLEb(st.tl)
\end{aligned}$$

The next programming task is the translation of the operations and functions of the specification into procedures and functions of the Ada program; assertions are included in the Ada code to document the relationship between the Ada functions and procedures and the VDM-SL functions and operations. The assertions are expressed as VDM-SL predicates relating the program to the specification via the retrieve functions.

10.5.2 Procedures and functions

The *INIT* operation is implemented by an *Init* procedure.

```
procedure Init(st: out Symbol_Table) is
begin
   st := new Symbol_Table_Element'(null, null);
   -- retr-SYMBOL-TABLEb(st) = [nil]
end Init;
```

The *OPEN-SCOPE* operation prefixes *st* with an empty table. The *Init* procedure is used to create an empty symbol table which is concatenated with *st*.

```
procedure Open_Scope is
   -- ext wr st: Symbol_Table;
   empty: Symbol_Table;
begin
   Init(empty);
   empty.tl := st;
   st := empty;
   -- retr-SYMBOL-TABLEb(st) = [nil] ⁀ retr-SYMBOL-TABLEb(st↼)
end Open_Scope;
```

The *CLOSE-SCOPE* operation is more difficult to implement than *OPEN-SCOPE*. Assume a *Dispose* procedure which may be used to deallocate the store allocated by the new function.

```
procedure Close_Scope is
   -- ext wr st: Symbol_Table;
   alias: Symbol_Table;
begin
   -- st ≠ null
   alias := st;
   st := st.tl;
   Dispose_Table(alias.hd);
   Dispose(alias);
   -- retr-SYMBOL-TABLEb(st) = tl retr-SYMBOL-TABLEb(st↼)
end Close_Scope;
```

The procedures which dispose of tables and lists are straightforward and are omitted. The implementation of the *ADD* operation uses a sub-procedure to update the head of the *st* symbol table.

```
procedure Add(id: in Identifier; tr: in Type_Record) is
   -- ext wr st: Symbol_Table;
begin  -- st ≠ null
   Insert(st.hd, id, tr)
   -- retr-SYMBOL-TABLEb(st) =
   --     [insert(retr-SYMBOL-TABLEb(hd st↼), id, tr)] ⁀
   --         tl retr-SYMBOL-TABLEb(st↼)
end Add;
```

The *Insert* procedure achieves the effect of the *insert* function by performing an in-place update on the table.

```
procedure Insert(t: in out Table; id: in Identifier; tr: in Type_Record) is
begin
   if t = null then
      t := new Node'(id, new Type_Record_Element'(tr, null), null, null);
      -- retr-TABLEb(t) = mk-TABLEb(id, [tr], nil, nil)
   elsif id < t.key then Insert(t.left, id, tr);
   elsif t.key < id then Insert(t.right, id, tr);
   else t.value := new Type_Record_Element'(tr, t.value);
      -- retr-TABLEb(t) =
      --     mk-TABLEb(id, [tr] ⁀ retr-TYPE-RECORDSb(t.value↼),
      --         retr-TABLEb(t.left↼), retr-TABLEb(t.right↼))
   end if;
end Insert;
```

The implementation of the *is-in* function makes a small optimisation to the algorithm suggested by the specification of the function by short-circuiting the evaluation of the disjunction once the identifier has been found.

```
function Is_In(id: in Identifier; st: in Symbol_Table) return Boolean is
   -- Is_In(id, st) = is-in(id, retr-SYMBOL-TABLEb(st))
begin  -- st ≠ null
   if st.tl = null then return In_Table(id, st.hd);
   elsif In_Table(id, st.hd) then return true;
   else return Is_In(id, st.tl)
   end if;
end Is_In;
```

The *In_Table* function searches the table for an identifier using the algorithm suggested by the specification.

```
function In_Table(id: in Identifier; t: in Table) return Boolean is
  -- In_Table(id, t) = in-table(id, retr-TABLEb(t))
begin
  if t = null then return false;
  elsif id < t.key then return In_Table(id, t.left);
  elsif t.key < id then return In_Table(id, t.right);
  else return true;
  end if;
end In_Table;
```

When given a linked list of only one element, the *Get_Types* function will use the *Types_Of* function to search the tree for the node containing the identifier *id*. When given a longer list, the function will use the *Types_Of* function to search the head and search the tail recursively. The resulting lists are concatenated.

```
function Get_Types(id: in Identifier;
                   st: in Symbol_Table) return Type_Records is
  -- retr-TYPE-RECORDSb(Get_Types(id, st)) =
  --     get-types(id, retr-SYMBOL-TABLEb(st))
  first, last: Type_Records;
begin  -- st ≠ null
  if st.tl = null then return Types_Of(id, st.hd);
  else
    first := Types_Of(id, st.hd);
    last := Get_Types(id, st.tl);
    Concatenate(first, last);
    return first;
  end if;
end Get_Types;
```

The *Types_Of* function searches the table using the algorithm used by the *In_Table* function. It is important not to alter the table t. Consequently, the values which are found to be associated with *id* are copied using the *CopyOf* function.

```
function Types_Of(id: in Identifier; t: in Table) return Type_Records is
  -- retr-TYPE-RECORDSb(Types_Of(id, t)) = types-of(id, retr-TABLEb(t))
begin
  if t = null then return null;
  elsif id < t.key then return Types_Of(id, t.left);
  elsif t.key < id then return Types_Of(id, t.right);
  else return CopyOf(t.value);
  end if;
end Types_Of;
```

The function which copies a list is straightforward and is omitted.

To complete the implementation of the *Get_Types* function a procedure is required which concatenates two linked lists of type records. Concatenating an empty list and another is simple. They may be connected by assignment. If the first list is non-empty, a recursive invocation of the procedure is used.

```
procedure Concatenate(s1: in out Type_Records; s2: in Type_Records) is
begin
   if s1 = null then s1 := s2; else Concatenate(s1.tl, s2); end if;
   -- retr-TYPE-RECORDSb(s1) =
   --     retr-TYPE-RECORDSb(↼s1) ⌒ retr-TYPE-RECORDSb(↼s2)
end Concatenate;
```

This completes the implementation of the specification.

10.6 Proofs about this implementation

In this section, proofs are provided to illustrate that this implementation satisfies the specification with respect to the retrieve functions.

10.6.1 Adequacy of the retrieve functions

The retrieve functions are simple enough to be taken as obviously adequate. In fact, the retrieve functions for the *TYPE-RECORDS* and *TABLE* types are total bijective functions. Thus, the concrete data space is no larger than the abstract data space. This is further proof that the data reification work being undertaken in this design step is slight.

10.6.2 Operation and function satisfaction proofs

Formal proofs that the procedures and functions of the implementation satisfy the operations and functions of the specification would require use of axiomatic proof rules for recursive procedures and functions and the verification of the assertions in the program code. Only an outline of the proofs is provided.

10.6.3 The *Init* and *Open_Scope* procedures

From the definition of the *retr-SYMBOL-TABLEb* function, *st* would be mapped to the symbol table [nil] as required. This completes the proof of the *Init* procedure. A proof of the *Open_Scope* procedure is similar.

10.6.4 The *Close_Scope* procedure

The *Close_Scope* procedure initially sets up an alias for the symbol table *st*. The names *alias* and *st* now refer to the same variable. The *st* variable is then altered to refer to the tail of its old value. The table at the head of the old symbol table is then lost. Finally, the alias of the old symbol table is lost.

10.6.5 The *Add* and *Insert* procedures

A proof of correctness of the *Add* and *Insert* procedures should establish the following.

1. Given a table which does not contain *id*, the *Insert* procedure will create a sub-tree which is mapped to $mk\text{-}TABLEb(id, [tr], \mathsf{nil}, \mathsf{nil})$ by *retr-TABLEb*.

2. Given a table containing id, the value at the relevant node is updated to correspond to the $[tr] \curvearrowright$ "old value" case.

10.6.6 The *Is_In* and *In_Table* functions

By basing the structure of the implementations of the *is-in* and *in-table* functions closely on the structure of the VDM-SL specifications, termination of the Ada functions is ensured by the totality of the VDM-SL functions. It only remains to show that the functions return the correct results.

Consider the *Is_In* function. For a non-empty symbol table st, *Is_In* is defined as follows.

$$Is\_In(id, st) = \begin{cases} In\_Table(id, st.hd) & \text{if } st.tl = null \\ true & \text{if } st.tl \neq null \text{ and } In\_Table(id, st.hd) \\ Is\_In(id, st.tl) & \text{otherwise} \end{cases}$$

This function expression may be simplified by folding the last two cases together.

$$Is\_In(id, st) = \begin{cases} In\_Table(id, st.hd) & \text{if } st.tl = null \\ In\_Table(id, st.hd) \textsf{ or } Is\_In(id, st.tl) & \text{otherwise} \end{cases}$$

This simple reorganization reveals the correspondence between the implementation and the specification. A similar association may be revealed for the *In_Table* function and its specification.

10.6.7 The *Get_Types* function

Proofs of the *Get_Types* and *Types_Of* functions are similar to proofs of *Is_In* and *In_Table* and are omitted. The *Concatenate* procedure is considered.

In the *Concatenate* procedure no copying of linked data structures is required. If s_1 is the empty list then s_1 is made an alias for s_2. If not, the lists $s_1.tl$ and s_2 are concatenated.

$$\begin{array}{l}
\textsf{if } s_1 = null \textsf{ then } s_1 := s_2; \\
\quad -- \; retr\text{-}TYPE\text{-}RECORDSb(s_1) = \\
\quad -- \quad\; retr\text{-}TYPE\text{-}RECORDSb(\overleftarrow{s_1}) \curvearrowright retr\text{-}TYPE\text{-}RECORDSb(\overleftarrow{s_2}) \\
\textsf{else } -- \; s_1 \neq null \\
\quad Concatenate(s_1.tl, s_2); \\
\quad -- \; s_1.hd = \overleftarrow{s_1.hd} \wedge retr\text{-}TYPE\text{-}RECORDSb(s_1.tl) = \\
\quad -- \quad\; retr\text{-}TYPE\text{-}RECORDSb(\overleftarrow{s_1.tl}) \curvearrowright retr\text{-}TYPE\text{-}RECORDSb(\overleftarrow{s_2}) \\
\textsf{end if};
\end{array}$$

This completes the proofs of the implementation. The definitions now need only be placed in declare-before-use order and sub-programs nested within sub-programs where appropriate to clarify the structure of the program.

10.7 Conclusions

This chapter is a summary of a larger development. Two levels of the specification are omitted from the start. The first used relations to represent a table. This is the most obvious model for overloading. For example, the following table records the fact the x has been declared as a Z schema and a Z function.

$$\{x \mapsto Schema, x \mapsto Function\}$$

Regrettably, the `map` type of VDM-SL does not represent mathematical mappings. A VDM-SL `map` can represent only a many-to-one mapping. Consequently, an alternative model was required to represent overloading. A `map` to sets of types was used.

$$\{x \mapsto \{Schema, Function\}\}$$

The first data reification replaced the set by a sequence. The discussion in this chapter begins from that point.

Obviously, the reader may feel that the author's ignorance of VDM-SL has led him to waste time writing the first draft of the specification and that consequently he should expect very little sympathy. However—on behalf of the VDM-SL *illiterati*—it might be said that the use of the name "`map`" to denote mathematical functions was poorly chosen.

The novel features of this development are:

- The data reification of types which are used in the definition of other types. Retrieve functions are needed for the super-type as well as the sub-type.
- The "language reification" step which uses a data reification across a language boundary to help ensure that the program code satisfies the lowest level specification.

For ease of readability, some of the rigorous natural deduction style proofs produced during this development have been omitted. It is essential to discharge such proof obligations in a VDM development but in order to bring the development work to a wider audience the proofs should be summarized or rewritten in natural language.

Chapter 11

An Algebraic Development of a Z Table Builder

This chapter presents an algebraic specification of a Z 'table builder' which fills a symbol table with definitions. The table builder also performs the final pre-processing before the translation from Z to VDM-SL: it separates definitions of functions and relations which have been grouped using one of the Z 'boxes'.

11.1 Algebraic specification style

The algebraic specifications presented here are expressed in a concrete syntax similar to that used in the VDM-SL specification language. This is done to ease the task of relating these specifications to the earlier model-oriented specifications. The change of concrete syntax aside, the language used is a simple algebraic language defined in (Sannella & Wirsing, 1983). The language resembles very high-level programming languages such as OBJ (Goguen & Tardo, 1979) but the specifications are not executable. The language uses conditional equations and permits identifier overloading—several functions may be given the same name if the signatures of the functions are different. Also, a function may have the same name as a type—here, in keeping with the literature on algebraic specifications, called a *sort*.

To simplify this specification, all of the functions defined will be made total by supplying default values for the error cases. A more sophisticated approach to error handling would obscure the clarity of the specification.

11.2 Specification

11.2.1 Basic sorts

Several sorts are taken as *given*. These are: *identifier*, *type*, *predicate*, *schema-exp* and *non-definition*. Declarations are defined to be composed of a sequence of names and a sort. VDM-like selector functions and a *mk*-function are defined for these. It will be necessary to construct the set of free variables of a predicate but this function is not specified since it is sufficiently well understood.

$DECLARATION$
sorts $identifier$
$type$
$declaration$
opns $names\colon declaration \rightarrow identifier^*$
$sort\colon declaration \rightarrow type$
$mk\text{-}declaration\colon identifier^* \times type \rightarrow declaration$
eqns $\forall\, d\colon declaration \cdot$
$mk\text{-}declaration(names(d), sort(d)) = d$

$PREDICATE$
sorts $identifier$
$predicate$
opns $free\text{-}vars\colon predicate \rightarrow identifier\text{-}\mathsf{set}$
$free\text{-}vars\colon predicate^* \rightarrow identifier\text{-}\mathsf{set}$
eqns $\forall\, p\colon predicate, s\colon predicate^* \cdot$
"$free\text{-}vars(p)$ is the set of free variables of p"

$free\text{-}vars(s) = \{\,\}$ if $s = [\,]$
$free\text{-}vars(s) = free\text{-}vars(\mathsf{hd}\, s) \cup free\text{-}vars(\mathsf{tl}\, s)$ if $s \neq [\,]$

11.2.2 Composite sorts

A schema text consists of a sequence of declarations and a sequence of predicates. A schema box is a named and possibly parameterised schema text. An empty schema text is defined. This contains no declarations and no predicates. An empty schema box is then defined. This is of unspecified name and without parameters and has an empty schema text as its body.

Notation: A '_ + _' operator is used to combine two algebras to make one with all the sorts, operations and equations of its left and right arguments. The keyword enrich is used to introduce the addition of further sorts, operations or equations to an existing algebra.

$SCHEMA\text{-}TEXT$
enrich $DECLARATION + PREDICATE$ by
sorts $schema\text{-}text$
opns $dec\text{-}list\colon schema\text{-}text \rightarrow declaration^*$
$pred\text{-}list\colon schema\text{-}text \rightarrow predicate^*$
$mk\text{-}schema\text{-}text\colon declaration^* \times predicate^* \rightarrow schema\text{-}text$
$empty\colon \rightarrow schema\text{-}text$
eqns $\forall\, s\colon schema\text{-}text \cdot$
$mk\text{-}schema\text{-}text(dec\text{-}list(s), pred\text{-}list(s)) = s$

$empty = mk\text{-}schema\text{-}text([\,], [\,])$

$SCHEMA\text{-}BOX$
enrich $SCHEMA\text{-}TEXT$ by
sorts $schema\text{-}box$
opns $name: schema\text{-}box \rightarrow identifier$
$parameters: schema\text{-}box \rightarrow identifier^*$
$body: schema\text{-}box \rightarrow schema\text{-}text$
$mk\text{-}schema\text{-}box: identifier \times identifier^* \times schema\text{-}text \rightarrow schema\text{-}box$
$empty: \rightarrow schema\text{-}box$
eqns $\forall s: schema\text{-}box \cdot$
$mk\text{-}schema\text{-}box(name(s), parameters(s), body(s)) = s$
$parameters(empty) = [\,]$
$body(empty) = empty$

11.2.3 Processing schema texts

A schema text consists of a sequence of declarations and a sequence of predicates. The following operations generate a sequence of schema texts from a single schema text by removing the first declaration and the relevant predicates, forming a schema text from these and proceeding in this way until no declarations remain.

There follows the problem of determining which predicates are relevant to a declaration. This is not difficult. If the identifier of the declaration occurs as a free variable of the predicate then the predicate is relevant: otherwise it is not.

Consider the processing of the following Z schema text. The variables defined are placed in separate schema texts. Given this schema text:

$x, y: \mathbb{N}$
$s: \mathbb{P}\,\mathbb{N}$

$x < 100$
$y < 200$
$x + y < 250$
$s \neq \varnothing$

the 'table builder' will produce the following three schema texts. Note that variable y mentioned in the first schema text is a forward reference to the variable y of type $\mathbb{N}$ defined in the second schema text.

$x: \mathbb{N}$

$x < 100$
$x + y < 250$

$y: \mathbb{N}$

$y < 200$
$x + y < 250$

$s: \mathbb{P}\,\mathbb{N}$

$s \neq \varnothing$

Before defining functions for processing schema texts, definitions are given for auxiliary functions which process declarations and sequences of declarations.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{PROCESS\text{-}DECLARATION}} \\
\textsf{enrich} & \mathit{DECLARATION}\ \textsf{by} \\
\textsf{opns} & \mathit{first}, \mathit{rest}\colon \mathit{declaration} \to \mathit{declaration} \\
\textsf{eqns} & \forall\, d\colon \mathit{declaration} \cdot \\
& \quad \mathit{first}(d) = d \qquad \textsf{if } \mathit{names}(d) = [\,] \\
& \quad \mathit{first}(d) = \mathit{mk\text{-}declaration}([\textsf{hd}\ \mathit{names}(d)], \mathit{sort}(d)) \qquad \textsf{if } \mathit{names}(d) \neq [\,] \\
& \quad \mathit{rest}(d) = d \qquad \textsf{if } \mathit{names}(d) = [\,] \\
& \quad \mathit{rest}(d) = \mathit{mk\text{-}declaration}(\textsf{tl}\ \mathit{names}(d), \mathit{sort}(d)) \qquad \textsf{if } \mathit{names}(d) \neq [\,]
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{PROCESS\text{-}DECLARATION\text{-}SEQ}} \\
\textsf{enrich} & \mathit{PROCESS\text{-}DECLARATION}\ \textsf{by} \\
\textsf{opns} & \mathit{first}, \mathit{rest}\colon \mathit{declaration}^* \to \mathit{declaration}^* \\
\textsf{eqns} & \forall\, s\colon \mathit{declaration}^* \cdot \\
& \quad \mathit{first}(s) = [\,] \qquad \textsf{if } s = [\,] \\
& \quad \mathit{first}(s) = [\mathit{first}(\textsf{hd}\ s)] \qquad \textsf{if } s \neq [\,] \\
& \quad \mathit{rest}(s) = [\,] \qquad \textsf{if } s = [\,] \\
& \quad \mathit{rest}(s) = \textsf{tl}\ s \qquad \textsf{if } s \neq [\,] \wedge \mathit{names}(\mathit{rest}(\textsf{hd}\ s)) = [\,] \\
& \quad \mathit{rest}(s) = [\mathit{rest}(\textsf{hd}\ s)] \frown \textsf{tl}\ s \qquad \textsf{if } s \neq [\,] \wedge \mathit{names}(\mathit{rest}(\textsf{hd}\ s)) \neq [\,]
\end{array}
$$

The following functions process schema texts.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{PROCESS\text{-}SCHEMA\text{-}TEXT}} \\
\textsf{enrich} & \mathit{SCHEMA\text{-}TEXT} + \mathit{PROCESS\text{-}DECLARATION\text{-}SEQ}\ \textsf{by} \\
\textsf{opns} & \mathit{first}, \mathit{rest}\colon \mathit{schema\text{-}text} \to \mathit{schema\text{-}text} \\
& \mathit{filter}\colon \mathit{predicate}^* \times \mathit{identifier}\text{-}\textsf{set} \to \mathit{predicate}^* \\
\textsf{eqns} & \forall\, s\colon \mathit{schema\text{-}text}, p\colon \mathit{predicate}^*, \mathit{ids}\colon \mathit{identifier}\text{-}\textsf{set} \cdot \\
& \quad \textsf{let } \mathit{decs} = \mathit{first}(\mathit{dec\text{-}list}(s))\ \textsf{in} \\
& \quad \textsf{let } \mathit{name} = \textsf{hd}\ \mathit{names}(\textsf{hd}\ \mathit{decs})\ \textsf{in} \\
& \quad \mathit{first}(s) = \mathit{empty} \qquad \textsf{if } \mathit{dec\text{-}list}(s) = [\,] \\
& \quad \mathit{first}(s) = \mathit{mk\text{-}schema\text{-}text}(\mathit{decs}, \mathit{filter}(\mathit{pred\text{-}list}(s), \{\mathit{name}\})) \\
& \qquad\qquad \textsf{if } \mathit{dec\text{-}list}(s) \neq [\,] \\
& \quad \mathit{rest}(s) = \mathit{empty} \qquad \textsf{if } \mathit{dec\text{-}list}(s) = [\,] \\
& \quad \mathit{rest}(s) = \mathit{mk\text{-}schema\text{-}text}(\mathit{rest}(\mathit{dec\text{-}list}(s)), \mathit{pred\text{-}list}(s)) \\
& \qquad\qquad \textsf{if } \mathit{dec\text{-}list}(s) \neq [\,] \\
& \quad \mathit{filter}(p, \mathit{ids}) = [\,] \qquad \textsf{if } p = [\,] \\
& \quad \mathit{filter}(p, \mathit{ids}) = \mathit{filter}(\textsf{tl}\ p, \mathit{ids}) \qquad \textsf{if } p \neq [\,] \wedge \mathit{free\text{-}vars}(\textsf{hd}\ p) \cap \mathit{ids} = \{\,\} \\
& \quad \mathit{filter}(p, \mathit{ids}) = [\textsf{hd}\ p] \frown \mathit{filter}(\textsf{tl}\ p, \mathit{ids}) \\
& \qquad\qquad \textsf{if } p \neq [\,] \wedge \mathit{free\text{-}vars}(\textsf{hd}\ p) \cap \mathit{ids} \neq \{\,\}
\end{array}
$$

11.2.4 Additional sorts

Additional sorts are now defined. A partial definition is given for a *schema-exp* sort which denotes expressions of the Z schema calculus. A constant *empty* of this sort is defined but no constructors are provided for the *schema-exp* sort since they are not needed in this specification.

$SCHEMA\text{-}EXP$
sorts $schema\text{-}exp$
opns $\vdots$
$empty \colon \to schema\text{-}exp$
eqns $\vdots$

Axiomatic definitions are parameterised schema texts. A suitable sort is now defined.

$AXIOMATIC\text{-}DEFN$
enrich $SCHEMA\text{-}TEXT$ by
sorts $axiomatic\text{-}defn$
opns $parameters \colon axiomatic\text{-}defn \to identifier^*$
$schema\text{-}text \colon axiomatic\text{-}defn \to schema\text{-}text$
$dec\text{-}list \colon axiomatic\text{-}defn \to declaration^*$
$pred\text{-}list \colon axiomatic\text{-}defn \to predicate^*$
$mk\text{-}axiomatic\text{-}defn \colon identifier^* \times schema\text{-}text \to axiomatic\text{-}defn$
$empty \colon \to axiomatic\text{-}defn$
eqns $\forall\, a \colon axiomatic\text{-}defn \cdot$

$$mk\text{-}axiomatic\text{-}defn(parameters(a), schema\text{-}text(a)) = a$$

$$schema\text{-}text(a) = mk\text{-}schema\text{-}text(dec\text{-}list(a), pred\text{-}list(a))$$

$$empty = mk\text{-}axiomatic\text{-}defn([\,], empty)$$

11.2.5 Processing axiomatic definitions

The principal requirement in processing an axiomatic definition is to 'unwind' a single definition to produce a sequence of smaller ones. An *unwind* function is provided for this purpose as are versions of the *first* and *rest* functions for axiomatic definitions. A *filter* function is used to trim the parameter list to just those names which are needed for each reduced definition.

$PROCESS\text{-}AXIOMATIC\text{-}DEFN$
enrich $AXIOMATIC\text{-}DEFN + PROCESS\text{-}SCHEMA\text{-}TEXT$ by
opns $first, rest \colon axiomatic\text{-}defn \to axiomatic\text{-}defn$
$unwind \colon axiomatic\text{-}defn \to axiomatic\text{-}defn^*$
$filter \colon identifier^* \times identifier\text{-}\mathsf{set} \to identifier^*$
eqns $\forall\, a \colon axiomatic\text{-}defn,\ ids_1 \colon identifier^*,\ ids_2 \colon identifier\text{-}\mathsf{set} \cdot$

$$\begin{array}{ll}
\mathsf{let}\ f = first(schema\text{-}text(a))\ \mathsf{in} & \\
first(a) = mk\text{-}axiomatic\text{-}defn([\,], empty) & \mathsf{if}\ dec\text{-}list(a) = [\,] \\
first(a) = mk\text{-}axiomatic\text{-}defn(filter(parameters(a), & \\
\quad free\text{-}vars(pred\text{-}list(f))), f) & \mathsf{if}\ dec\text{-}list(a) \neq [\,] \\
rest(a) = empty & \mathsf{if}\ dec\text{-}list(a) = [\,] \\
rest(a) = mk\text{-}axiomatic\text{-}defn(parameters(a), rest(schema\text{-}text(a))) & \\
 & \mathsf{if}\ dec\text{-}list(a) \neq [\,] \\
unwind(a) = [\,] & \mathsf{if}\ dec\text{-}list(a) = [\,] \\
unwind(a) = [first(a)] \frown unwind(rest(a)) & \mathsf{if}\ dec\text{-}list(a) \neq [\,]
\end{array}$$

$$
\begin{aligned}
&\mathit{filter}(ids_1, ids_2) = [\,] && \textsf{if } ids_1 = [\,]\\
&\mathit{filter}(ids_1, ids_2) = \mathit{filter}(\textsf{tl}\, ids_1, ids_2) && \textsf{if } ids_1 \neq [\,] \land \textsf{hd}\, ids_1 \notin ids_2\\
&\mathit{filter}(ids_1, ids_2) = [\textsf{hd}\, ids_1] \frown \mathit{filter}(\textsf{tl}\, ids_1, ids_2) &&\\
& && \textsf{if } ids_1 \neq [\,] \land \textsf{hd}\, ids_1 \in ids_2
\end{aligned}
$$

11.2.6 Structured sorts

Schema boxes, schema expressions and axiomatic definitions are now combined to form a larger sort—*definition*—which has three constructor functions and a classification function to determine which of the three selector functions to use.

DEFINITION
enrich *AXIOMATIC-DEFN* + *SCHEMA-EXP* + *SCHEMA-BOX* by
sorts *definition*
definition-type
opns *an-axiomatic-defn*,
a-schema-exp,
a-schema-box,
$\mathit{empty}\colon \to \mathit{definition\text{-}type}$
$\mathit{mk\text{-}definition}\colon \mathit{axiomatic\text{-}defn} \to \mathit{definition}$
$\mathit{mk\text{-}definition}\colon \mathit{schema\text{-}exp} \to \mathit{definition}$
$\mathit{mk\text{-}definition}\colon \mathit{schema\text{-}box} \to \mathit{definition}$
$\mathit{class}\colon \mathit{definition} \to \mathit{definition\text{-}type}$
$\mathit{axiomatic\text{-}defn}\colon \mathit{definition} \to \mathit{axiomatic\text{-}defn}$
$\mathit{schema\text{-}exp}\colon \mathit{definition} \to \mathit{schema\text{-}exp}$
$\mathit{schema\text{-}box}\colon \mathit{definition} \to \mathit{schema\text{-}box}$
$\mathit{empty}\colon \to \mathit{definition}$
eqns $\forall\, ad\colon \mathit{axiomatic\text{-}defn}, se\colon \mathit{schema\text{-}exp}, sb\colon \mathit{schema\text{-}box}\ \cdot$

$$
\begin{aligned}
&\mathit{class}(\mathit{mk\text{-}definition}(ad)) = \mathit{an\text{-}axiomatic\text{-}defn}\\
&\mathit{class}(\mathit{mk\text{-}definition}(se)) = \mathit{a\text{-}schema\text{-}exp}\\
&\mathit{class}(\mathit{mk\text{-}definition}(sb)) = \mathit{a\text{-}schema\text{-}box}\\
&\mathit{class}(\mathit{empty}) = \mathit{empty}\\[1em]
&\mathit{axiomatic\text{-}defn}(\mathit{mk\text{-}definition}(ad)) = ad\\
&\mathit{axiomatic\text{-}defn}(\mathit{mk\text{-}definition}(se)) = \mathit{empty}\\
&\mathit{axiomatic\text{-}defn}(\mathit{mk\text{-}definition}(sb)) = \mathit{empty}\\
&\mathit{axiomatic\text{-}defn}(\mathit{empty}) = \mathit{empty}\\[1em]
&\mathit{schema\text{-}exp}(\mathit{mk\text{-}definition}(ad)) = \mathit{empty}\\
&\mathit{schema\text{-}exp}(\mathit{mk\text{-}definition}(se)) = se\\
&\mathit{schema\text{-}exp}(\mathit{mk\text{-}definition}(sb)) = \mathit{empty}\\
&\mathit{schema\text{-}exp}(\mathit{empty}) = \mathit{empty}\\[1em]
&\mathit{schema\text{-}box}(\mathit{mk\text{-}definition}(ad)) = \mathit{empty}\\
&\mathit{schema\text{-}box}(\mathit{mk\text{-}definition}(se)) = \mathit{empty}\\
&\mathit{schema\text{-}box}(\mathit{mk\text{-}definition}(sb)) = sb\\
&\mathit{schema\text{-}box}(\mathit{empty}) = \mathit{empty}
\end{aligned}
$$

The following algebra introduces a sort for non-definitions. These are Z phrases which are not processed by the table builder. They are grouped together for convenience of specification only.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{NON-DEFINITION}} \\
\textsf{sorts} & \textit{non-definition} \\
\textsf{opns} & \quad\vdots \\
 & \textit{empty}\colon \to \textit{non-definition} \\
 & \quad\vdots \\
\textsf{eqns} & \quad\vdots
\end{array}
$$

A *node* sort is now defined to allow non-definitions and definitions to be combined.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{NODE}} \\
\textsf{enrich} & \textit{DEFINITION} + \textit{NON-DEFINITION}\ \textsf{by} \\
\textsf{sorts} & \textit{node} \\
 & \textit{node-type} \\
\textsf{opns} & \textit{a-definition}, \\
 & \textit{a-non-definition}\colon \to \textit{node-type} \\
 & \textit{mk-node}\colon \textit{definition} \to \textit{node} \\
 & \textit{mk-node}\colon \textit{non-definition} \to \textit{node} \\
 & \textit{class}\colon \textit{node} \to \textit{node-type} \\
 & \textit{definition}\colon \textit{node} \to \textit{definition} \\
 & \textit{non-definition}\colon \textit{node} \to \textit{non-definition} \\
\textsf{eqns} & \forall\, d\colon \textit{definition}, \textit{nd}\colon \textit{non-definition}\ \cdot \\
 & \qquad \textit{class}(\textit{mk-node}(d)) = \textit{a-definition} \\
 & \qquad \textit{class}(\textit{mk-node}(\textit{nd})) = \textit{a-non-definition} \\
 & \\
 & \qquad \textit{definition}(\textit{mk-node}(d)) = d \\
 & \qquad \textit{definition}(\textit{mk-node}(\textit{nd})) = \textit{empty} \\
 & \\
 & \qquad \textit{non-definition}(\textit{mk-node}(d)) = \textit{empty} \\
 & \qquad \textit{non-definition}(\textit{mk-node}(\textit{nd})) = \textit{nd}
\end{array}
$$

11.2.7 The symbol table

The symbol table is specified in Chapter 10 but a partial re-specification is given here.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{TABLE}} \\
\textsf{sorts} & \textit{table} \\
\textsf{opns} & \quad\vdots \\
 & \textit{add-table}\colon \textit{table} \times \textit{node} \to \textit{table} \\
 & \textit{add-table}\colon \textit{table} \times \textit{node}^* \to \textit{table} \\
 & \quad\vdots \\
\textsf{eqns} & \quad\vdots
\end{array}
$$

11.2.8 The state of the system

The system consists of input and output sequences of nodes and a symbol table.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{STATE}} \\
\mathsf{enrich} & \mathit{NODE} + \mathit{TABLE}\ \mathsf{by} \\
\mathsf{sorts} & \mathit{state} \\
\mathsf{opns} & \mathit{input}, \\
 & \mathit{output}\colon \mathit{state} \to \mathit{node}^* \\
 & \mathit{table}\colon \mathit{state} \to \mathit{table} \\
 & \mathit{mk\text{-}state}\colon \mathit{node}^* \times \mathit{node}^* \times \mathit{table} \to \mathit{state} \\
\mathsf{eqns} & \forall\, s\colon \mathit{state}\ \cdot \\
 & \quad \mathit{mk\text{-}state}(\mathit{input}(s), \mathit{output}(s), \mathit{table}(s)) = s
\end{array}
$$

11.2.9 The specification of the table builder

The principal function of the table builder is a function called *build-table* which is a function from state to state. It employs a function which processes a single node. This function in turn uses a function which processes a definition. Separate functions are provided for axiomatic definitions and for schema boxes. An auxiliary function called *wrap* generates a sequence of nodes from a sequence of axiomatic definitions.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{TABLE\text{-}BUILDER}} \\
\mathsf{enrich} & \mathit{STATE} + \mathit{PROCESS\text{-}AXIOMATIC\text{-}DEFN}\ \mathsf{by} \\
\mathsf{opns} & \mathit{build\text{-}table}\colon \mathit{state} \to \mathit{state} \\
 & \mathit{process\text{-}node}\colon \mathit{node} \times \mathit{state} \to \mathit{state} \\
 & \mathit{process\text{-}definition}\colon \mathit{definition} \times \mathit{state} \to \mathit{state} \\
 & \mathit{process\text{-}axiomatic\text{-}defn}\colon \mathit{axiomatic\text{-}defn} \times \mathit{state} \to \mathit{state} \\
 & \mathit{process\text{-}schema\text{-}box}\colon \mathit{schema\text{-}box} \times \mathit{state} \to \mathit{state} \\
 & \mathit{wrap}\colon \mathit{axiomatic\text{-}defn}^* \to \mathit{node}^* \\
\mathsf{eqns} & \forall\, s\colon \mathit{state}, n\colon \mathit{node}, d\colon \mathit{definition}, a\colon \mathit{axiomatic\text{-}defn}, \\
 & \quad sb\colon \mathit{schema\text{-}box}, as\colon \mathit{axiomatic\_defn}^*\ \cdot
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{let}\ pn = \mathit{process\text{-}node}(\mathsf{hd}\ \mathit{input}(s), s)\ \mathsf{in}} \\
\mathit{build\text{-}table}(s) = s & \mathsf{if}\ \mathit{input}(s) = [\,] \\
\multicolumn{2}{l}{\mathit{build\text{-}table}(s) = \mathit{mk\text{-}state}(\mathsf{tl}\ \mathit{input}(s), \mathit{output}(pn), \mathit{table}(pn))} \\
 & \mathsf{if}\ \mathit{input}(s) \neq [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{process\text{-}node}(n, s) = \mathit{mk\text{-}state}([\,], \mathit{output}(s) \frown [n], \mathit{table}(s)) \\
\hfill \mathsf{if}\ \mathit{class}(n) = \mathit{a\text{-}non\text{-}definition} \\
\mathit{process\text{-}node}(n, s) = \mathit{process\text{-}definition}(\mathit{definition}(n), s) \\
\hfill \mathsf{if}\ \mathit{class}(n) \neq \mathit{a\text{-}non\text{-}definition}
\end{array}
$$

$$
\begin{array}{l}
\mathit{process\text{-}definition}(d, s) = \mathit{process\text{-}axiomatic\text{-}defn}(\mathit{axiomatic\text{-}defn}(d), s) \\
\hfill \mathsf{if}\ \mathit{class}(d) = \mathit{an\text{-}axiomatic\text{-}defn} \\
\mathit{process\text{-}definition}(d, s) = \\
\quad \mathit{mk\text{-}state}([\,], \mathit{output}(s) \frown [\mathit{mk\text{-}node}(d)], \mathit{table}(s)) \\
\hfill \mathsf{if}\ \mathit{class}(d) = \mathit{a\text{-}schema\text{-}exp} \\
\mathit{process\text{-}definition}(d, s) = \mathit{process\text{-}schema\text{-}box}(\mathit{schema\text{-}box}(d), s)
\end{array}
$$

$$\text{if } class(d) = a\text{-}schema\text{-}box$$
$$process\text{-}definition(d, s) = s \qquad \text{if } class(d) = empty$$

$$\begin{array}{l}
\text{let } output = output(s) \curvearrowright wrap(unwind(a)) \text{ in} \\
\text{let } table = add\text{-}table(table(s), wrap(unwind(a))) \text{ in} \\
process\text{-}axiomatic\text{-}defn(a, s) = mk\text{-}state([\,], output, table)
\end{array}$$

$$\begin{array}{l}
\text{let } ns = [mk\text{-}node(mk\text{-}definition(sb))] \text{ in} \\
process\text{-}schema\text{-}box(sb, s) = \\
\quad mk\text{-}state([\,], output(s) \curvearrowright ns, add\text{-}table(table(s), ns))
\end{array}$$

$$\begin{array}{ll}
wrap(as) = [\,] & \text{if } as = [\,] \\
wrap(as) = [mk\text{-}node(mk\text{-}definition(\mathsf{hd}\ as))] \curvearrowright wrap(\mathsf{tl}\ as) & \text{if } as \neq [\,]
\end{array}$$

This completes the specification of the table builder.

11.3 Development

Many of the above functions are direct definitions of conditional recursive functions. The conditional function definitions have two important properties. First, the conditions are *disjoint.* That is, there is no way to ascribe values to the parameters of a function to make more than one of the conditions true. This reduces the danger of contradictions in the specification. Second, the conditions are *complementary.* That is, at least one of the conditions is true for all possible combinations of values of parameters. Taken in conjunction with the above, this ensures that exactly one of the definitions applies in each case. This reduces the danger of incompleteness of the specification.

Definitions such as *mk-declaration* and *mk-schema-text* describe cartesian product types which may be implemented using the record type of Ada. Declarations such as *mk-definition* and *mk-node* describe disjoint union types which may be implemented using the variant record facility of Ada.

Finally, the Ada language supports identifier overloading. This allows functions of the same name in the specification to be refined to functions of the same name in the implementation.

11.4 Conclusions

The language used in this development is much simpler than VDM-SL or Z because it does not have a basic library of pre-defined functions and datatypes which the user is obliged to master. In consequence, it is considerably easier to learn—although it should be noted that the language used here is not intended for large-scale software development, it is a simple algebraic language used to teach introductory algebraic specification concepts.

It is easy to underestimate the difficulty of learning a specification language. Even popular specification languages such as VDM and Z pose difficulties for many software engineers. Their more esoteric cousins may be beyond the grasp of many forever. Baroque logics, constructive mathematics and categorical systems appeal to certain kinds of minds. A facility with all these is not obligatory for a software engineer: it may not even be desirable since it may distract attention from more conventional but no less worthwhile research in more familiar application areas.

Correctness-oriented approaches to software development are not an end in themselves: merely tools to assist in the production of high quality software.

Algebraic specifications should appeal to software developers who are interested in correctness-oriented approaches to software development but who have misgivings about investing too much time in re-education. Programmers who are conversant with functional programming languages will recognize an affinity between functional language programs and algebraic specifications. The defining functions used above are mostly tail-recursive and suggest an implementation strategy. The constructors of the language are simpler than those of the Z schema calculus or the process combinators of CSP. This also simplifies the problems of implementation.

There is a penalty to be paid for simplicity. Algebraic specifications will frequently be longer than an equivalent model-oriented specification. The clarity of this specification would have been greatly improved if the VDM-SL declaration mechanism for composite objects had been adopted instead of attempting to simulate it using functions.

11.4.1 Specifications and programs

In texts on correctness-oriented specification and software development methods, it is often implicitly assumed that specification writing and software development occur in a totally arid environment. This is not the case in this project. A difficulty arises because the author is forced to aim data reifications and other refinements towards an existing data structure rather than follow the most appealing path.

11.4.2 Specification exemplars

When expounding a specification technique or notation, authors choose specification examples which demonstrate the economy or beauty of the approach. This behaviour is commendable: good ideas deserve good presentation. In non-exemplary use the formalisms do not always produce such aesthetically pleasing results. Non-exemplary specifications seem rather tawdry when compared with their highly polished rivals.

This specification should not be taken as a good example of the algebraic specification style. It is not even a good example of the use of formal notations for specification. Firstly, it is devoid of proofs of properties of the specification. A specification of high quality cannot be divorced from the provision of such proofs.

Chapter 12

A Z Development of a Z-to-VDM-SL Translator

This chapter describes the translation of Z texts into the VDM-SL notation. It includes abstract syntaxes for both Z and VDM-SL.

Two strategies are used to make the task of specifying the translation easier. Firstly, the abstract syntax for VDM-SL has been fitted to the shape of the existing Z abstract syntax. This results in a syntax for VDM-SL which seems unnatural but which adequately describes the language. Secondly, a core of expressions, predicates, identifiers and declarations of typed variables is identified and isolated to avoid re-specification of these structures.

These decisions suggest an order of presentation for the specification. Datatypes are presented first. Of these, the common core of the languages precedes the features found in one language and not the other. In pairs with the latter come specifications of the translation operations.

Because this specification contains an abstract syntax for the target language, it will also serve as the specification of the unparser of the translator.

12.1 Specification

12.1.1 Common syntax

This section contains abstract syntax definitions for the constructs which are common to both languages.

Identifiers

Identifiers in Z are more structured than in many other formal languages. Structured objects may be accessed for read-only use or read-write use. This is denoted by preceding the identifier with a qualifier. Variables may be marked as inputs or outputs, or as state-before or state-after. This is done by giving the variable an appropriate suffix. Variables may also be subscripted by a non-empty sequence of characters. Finally, a non-empty 'basename' sequence of alphanumeric characters is required for the identifier.

Notation: The *f*UZZ dialect of Z includes a mechanism for easily describing recursive structures such as trees. The following is an example of the use of this mechanism in what is termed a 'free type definition'. Although it does not add anything to the power of the language, the free type definition mechanism proves very useful and is used widely throughout this specification.

The free type notation allows us to introduce constructor functions for structured types. The constructor functions used below are *Subscript*, *Superscript* and *Id*. These functions are akin to the *mk*-functions of VDM.

$$Use ::= Read\_Only \mid Read\_Write \mid Unspecified$$

$$Mode ::= Input \mid Output \mid State\_Before \mid State\_After$$

$$\begin{aligned} Script ::= {} & Subscript\langle\!\langle \mathrm{seq}_1\ CHAR \rangle\!\rangle \\ \mid {} & Superscript\langle\!\langle \mathrm{seq}_1\ CHAR \rangle\!\rangle \\ \mid {} & No\_Script \end{aligned}$$

$$Identifier ::= Id\langle\!\langle Use \times \mathrm{seq}_1\ CHAR \times Script \times Mode \rangle\!\rangle$$

Expressions

Abstract syntax definitions for the classes of expression in the Z and VDM-SL languages are now given.

The Exp_0 type: At this point, two forward references to constructs defined later are given: these are the types *Constrained_Vars* and *Expression* and are described informally here.

The type *Constrained_Vars* is used to denote typed variables whose permitted values are optionally further restricted by a predicate. Constrained variables are termed 'schema texts' in Z.

The type *Expression* is used to denote any non-logical expression, e.g. $(a + b) * c$.

The Exp_0 type may be either a lambda or mu function expression or an expression.

$$\begin{aligned} Exp_0 ::= {} & A\_Lambda\_Function\langle\!\langle Constrained\_Vars \times Expression \rangle\!\rangle \\ \mid {} & A\_Mu\_Function\langle\!\langle Constrained\_Vars \times Expression \rangle\!\rangle \\ \mid {} & An\_Expression\langle\!\langle Expression \rangle\!\rangle \end{aligned}$$

Set expressions: Sets may be represented in many ways in Z. The members of a non-empty finite set many be enumerated, e.g. $\{1, 2, 3\}$. Sets may also be described by a predicate as in $\{\ x : \mathbb{N} \mid x \bmod 10 = 0\ \}$ or by an expression, e.g. $\{\ x : \mathbb{N} \cdot 10 * x\ \}$. The choice of which form to use is made in order to increase the clarity of the description of the set. Here, both sets denote the set of natural numbers which are divisible by 10. The expression form used in $\{\ x : \mathbb{N} \cdot 10 * x\ \}$ gives the more elegant description in this instance.

Empty sets are allowable set expressions. These are denoted by $\varnothing$ or by $\varnothing[Type]$ where it is necessary to disambiguate. The only pre-defined non-empty set in the Z language is the set of natural numbers, $\mathbb{N}$.

$$
\begin{aligned}
Set\_Exp ::= \ & A\_Set\_In\_Extension \langle\langle \mathrm{seq}_1\ Expression \rangle\rangle \\
\mid\ & A\_Set\_By\_Predicate \langle\langle Constrained\_Vars \rangle\rangle \\
\mid\ & A\_Set\_By\_Expression \langle\langle Constrained\_Vars \times Expression \rangle\rangle \\
\mid\ & An\_Empty\_Set \langle\langle Expression \rangle\rangle \\
\mid\ & The\_Empty\_Set \\
\mid\ & The\_Natural\_Numbers
\end{aligned}
$$

The Exp_{10} **type:** The Exp_{10} class of expressions describes literals of assorted types. An identifier with optional generic parameters is a valid Exp_{10} expression. Natural number, set, sequence, and tuple literals are also permitted. A parenthesized expression is also a valid Exp_{10} expression.

$$
\begin{aligned}
Exp_{10} ::= \ & An\_Identifier \langle\langle Identifier \times \mathrm{seq}\ Expression \rangle\rangle \\
\mid\ & A\_Natural\_Number \langle\langle \mathrm{seq}_1\ CHAR \rangle\rangle \\
\mid\ & A\_Set \langle\langle Set\_Exp \rangle\rangle \\
\mid\ & A\_Sequence \langle\langle \mathrm{seq}\ Expression \rangle\rangle \\
\mid\ & A\_Parenthesized\_Exp \langle\langle Expression \rangle\rangle \\
\mid\ & A\_Tuple \langle\langle Expression \times \mathrm{seq}_1\ Expression \rangle\rangle
\end{aligned}
$$

The Exp_9 **type:** The Exp_9 class of expressions describes postfix function applications. The postfix functions of Z all apply to relations. They are: inverse, $f^{\sim}$; transitive closure, f^{+}; reflexive transitive closure, f^{*}; and iteration, f^{n}.

$$
\begin{aligned}
Postfix\_Function ::= \ & Inverse \mid Transitive\_Closure \mid Reflexive\_Transitive\_Closure \\
\mid\ & Iteration \langle\langle Expression \rangle\rangle \\
Exp_9 ::= \ & An\_Exp_9 \langle\langle Exp_{10} \times \mathrm{seq}\ Postfix\_Function \rangle\rangle
\end{aligned}
$$

The Exp_8 **type:** The Exp_8 category of expressions describes function applications and relational image expressions.

$$
\begin{aligned}
Optional\_Param_8 ::= \ & Function\_Parameter\_List \langle\langle \mathrm{seq}_1\ Expression \rangle\rangle \\
\mid\ & Relational\_Image \langle\langle Exp_0 \rangle\rangle \\
\mid\ & No\_Parameters \\
Exp_8 ::= \ & An\_Exp_8 \langle\langle Exp_9 \times Optional\_Param_8 \rangle\rangle
\end{aligned}
$$

The Exp_7 **type:** The Exp_7 category defines prefix generic function application expressions.

$$
\begin{aligned}
Prefix\_Generic ::= \ & Powerset \mid Powerset_1 \mid Finite\_Subset \mid Finite\_Subset_1 \mid \\
& Identity\_Relation \mid Seq \mid Seq_1 \mid Distributed\_Union \mid Distributed\_Intersection \mid \\
& Dom \mid Ran \mid Cardinality \mid Distributed\_Concatenation \mid Unary\_Minus \mid \\
& Head \mid Tail \mid Last \mid Front \mid Reverse
\end{aligned}
$$

$$
Exp_7 ::= An\_Exp_7 \langle\langle \mathrm{seq}\ Prefix\_Generic \times Exp_8 \rangle\rangle
$$

The Exp_6 type: The Exp_6 category describes application of domain or range restriction or subtraction to a relation.

$$
\begin{aligned}
Infix\_Func_6 &::= Domain\_Restriction \mid Range\_Restriction \\
&\quad\mid\ Domain\_Subtraction \mid Range\_Subtraction \\
Link_6 &::= A\_Link_6 \langle\langle Exp_7 \times Infix\_Func_6 \rangle\rangle \\
Exp_6 &== \operatorname{seq}_1 Link_6
\end{aligned}
$$

The Exp_5 type: The Exp_5 category describes function overriding, e.g. $f \oplus g \oplus h$. Since there is only one operator with this precedence, it need not be represented explicitly.

$$Exp_5 == \operatorname{seq}_1 Exp_6$$

The Exp_4 type: The Exp_4 category describes expressions whose sub-expressions are combined by operators with the same precedence as the multiplication operator.

$$
\begin{aligned}
Infix\_Func_4 &::= Times \mid Div \mid Mod \mid Intersection \mid Relational\_Composition \\
&\quad\mid\ Forward\_Relational\_Composition \mid Sequence\_Restriction \\
Link_4 &::= A\_Link_4 \langle\langle Exp_5 \times Infix\_Func_4 \rangle\rangle \\
Exp_4 &== \operatorname{seq}_1 Link_4
\end{aligned}
$$

The Exp_3 type: The Exp_3 category describes expressions whose sub-expressions are combined by operators with the same precedence as the addition operator.

$$
\begin{aligned}
Infix\_Func_3 &::= Plus \mid Binary\_Minus \mid Union \mid Set\_Difference \\
&\quad\mid\ Sequence\_Concatenation \\
Link_3 &::= A\_Link_3 \langle\langle Exp_4 \times Infix\_Func_3 \rangle\rangle \\
Exp_3 &== \operatorname{seq}_1 Link_3
\end{aligned}
$$

The Exp_2 type: The Exp_2 category describes number ranges e.g. $1 \mathrel{..} 10$. As there is only one operator with this precedence—the double dot operator—it does not need to be represented explicitly. There will be at most two elements in the sequence.

$$Exp_2 == \operatorname{seq}_1 Exp_3$$

The Exp_1 type: The Exp_1 category describes maplets e.g. $x \mapsto y$. As there is only one operator with this precedence it need not be represented explicitly. There will again be at most two elements in the sequence.

$$Exp_1 == \operatorname{seq}_1 Exp_2$$

The *Expression* **type:** The *Expression* category used earlier is now defined. The *Expression* category permits representation of signatures of functions, e.g. $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$.

$$\begin{array}{lll} \mathit{Infix\_Generic} & ::= & \mathit{Total\_Function} \mid \mathit{Partial\_Function} \mid \mathit{Total\_Injection} \\ & \mid & \mathit{Partial\_Injection} \mid \mathit{Finite\_Injection} \mid \mathit{Total\_Surjection} \\ & \mid & \mathit{Partial\_Surjection} \mid \mathit{Bijection} \end{array}$$

$$\mathit{Expression\_Link} ::= \mathit{Exp\_Link}\langle\!\langle \mathrm{seq}_1\, \mathit{Exp}_1 \times \mathit{Infix\_Generic} \rangle\!\rangle$$

$$\mathit{Expression} == \mathrm{seq}_1\, \mathit{Expression\_Link}$$

This completes the abstract syntax for expressions.

Declarations

Now an abstract syntax is given for declarations of typed variables. There are two allowable classes of declarations, identifiers of structured objects containing constrained variables and lists of identifiers constrained by a type. A non-empty injective sequence—a sequence which cannot contain duplicate elements—is used to prohibit invalid declaration lists such as '$: \mathbb{N}$' and '$x, x: \mathbb{N}$'.

$$\begin{array}{ll} \mathit{Declaration} ::= & \mathit{Constrained\_Variables}\langle\!\langle \mathit{Identifier} \rangle\!\rangle \\ \mid & \mathit{Variables}\langle\!\langle \mathrm{iseq}_1\, \mathit{Identifier} \times \mathit{Expression} \rangle\!\rangle \end{array}$$

Logical expressions

A logical expression is either the name of a logical-valued variable or a relational expression consisting of two expressions and a relational operator, e.g. $x = 0$, or a parenthesized sequence of predicates. The type *Predicate* is another forward reference.

$$\begin{array}{l} \mathit{Rel\_Op} ::= \mathit{Equal} \mid \mathit{Member} \mid \mathit{Not\_Equal} \mid \mathit{Not\_Member} \mid \mathit{Subset} \mid \\ \quad \mathit{Proper\_Subset} \mid \mathit{Less\_Than} \mid \mathit{Greater\_Than} \mid \mathit{Less\_Than\_Or\_Equal} \mid \\ \quad \mathit{Greater\_Than\_Or\_Equal} \end{array}$$

$$\begin{array}{ll} \mathit{Logical\_Expression} ::= & \mathit{Logical\_Valued\_Var}\langle\!\langle \mathit{Identifier} \rangle\!\rangle \\ \mid & \mathit{Relational\_Exp}\langle\!\langle \mathit{Expression} \times \mathit{Rel\_Op} \times \mathit{Expression} \rangle\!\rangle \\ \mid & \mathit{Parenthesized\_Pred}\langle\!\langle \mathrm{seq}_1\, \mathit{Predicate} \rangle\!\rangle \end{array}$$

Predicates

$$\begin{array}{ll} \mathit{Negation} ::= & \mathit{A\_Logical\_Expression}\langle\!\langle \mathit{Logical\_Expression} \rangle\!\rangle \\ \mid & \mathit{Negated\_Logical\_Expression}\langle\!\langle \mathit{Logical\_Expression} \rangle\!\rangle \end{array}$$

$$\mathit{Conjunction} == \mathrm{seq}_1\, \mathit{Negation}$$

$$\mathit{Disjunction} == \mathrm{seq}_1\, \mathit{Conjunction}$$

$$\mathit{Implication} == \mathrm{seq}_1\, \mathit{Disjunction}$$

$$\mathit{Equivalence} == \mathrm{seq}_1\, \mathit{Implication}$$

Below is the definition of the *Predicate* type used earlier.

$$Quantifier ::= Universal \mid Existential \mid Unique$$

$$\begin{aligned} Predicate ::= {} & Quantified\_Predicate\langle\!\langle Quantifier \times \\ & \quad Constrained\_Vars \times Predicate\rangle\!\rangle \\ \mid {} & Unquantified\_Predicate\langle\!\langle Equivalence\rangle\!\rangle \end{aligned}$$

Constrained variables

Constrained variables are formed from a non-empty injective sequence of declarations and a sequence of predicates.

$$Constrained\_Vars ::= Constrained\langle\!\langle \mathrm{iseq}_1\, Declaration \times \mathrm{seq}\, Predicate\rangle\!\rangle$$

Introduce two projector functions for constrained variables, *decs* and *pred*.

$$\begin{array}{|l} decs : Constrained\_Vars \rightarrow \mathrm{iseq}_1\, Declaration \\ pred : Constrained\_Vars \rightarrow \mathrm{seq}\, Predicate \\ \hline \forall\, vars : Constrained\_Vars \cdot vars = Constrained\ (decs\ (vars), pred\ (vars)) \end{array}$$

In the author's opinion, this is an area where the VDM-SL notation is preferable. Structured datatypes are common both in specifications and in programs and deserve good notation for their representation. In VDM-SL, constructor *mk*-functions and projector functions are automatically defined for all structured types. In Z, only the constructor functions are pre-defined. This can lead to clumsiness in their use: it is often necessary to introduce dummy variables in order to use the constructor function to give a name to a component of the type. In VDM-SL, this is never necessary.

12.1.2 Specialized definitions

Definitions of typed variables and predicates are common to both the Z and VDM-SL languages. The principal differences between the languages are the interpretation and selection of predicates to constrain initial and final values of variables and the structuring facilities available for combining operations.

The above declarations are regarded as common to both languages but the following declarations use terminology from one language or the other. Prefixes *Z* and *VDM* are used to distinguish the identifiers.

The corresponding structures of the languages are defined in pairs. These definitions are then followed by the specification of the operation which translates the Z structure to the corresponding VDM-SL structure.

Schema expressions

The specification of the Z calculus for combining schemas now follows. VDM-SL has no direct equivalent of this feature: schema expressions are translated by expanding the expression using the definitions of the schema combinators to obtain a single schema.

$$Z\_Schema\_Prefix ::= Schema\_Negation \mid Schema\_Precondition$$

$$\begin{aligned} Z\_Prefixed\_Schema ::= {} & Prefixed\_Schema\langle\!\langle \mathrm{seq}\, Z\_Schema\_Prefix \times Identifier \times \\ & \quad \mathrm{seq}\, Expression\rangle\!\rangle \end{aligned}$$

$$\begin{aligned} Z\_Schema\_Hiding \; ::= \; & A\_Prefixed\_Schema \langle\!\langle Z\_Prefixed\_Schema \rangle\!\rangle \\ \mid \; & A\_Hidden\_Prefixed\_Schema \langle\!\langle Z\_Prefixed\_Schema \times \\ & \quad Z\_Prefixed\_Schema \rangle\!\rangle \end{aligned}$$

A schema expression can contain any number of schemas connected by equivalence, implication, disjunction, conjunction, composition and schema hiding. For example, the following is a schema definition which uses all the schema operators.

$$Z \mathrel{\widehat=} P \Leftrightarrow Q \Rightarrow R \vee S \wedge T \wedge U \mathbin{;} V \setminus W \setminus X \mathbin{;} Y$$

The following fully parenthesized form illustrates the precedence of the operators used above.

$$Z \mathrel{\widehat=} (P \Leftrightarrow (Q \Rightarrow (R \vee ((S \wedge T) \wedge ((U \mathbin{;} ((V \setminus W) \setminus X)) \mathbin{;} Y)))))$$

Because there is only one operator of each precedence, the operators can be omitted in the abstract syntax leaving only nesting of sequences to reflect the meaning of the expression.

$$Z \mathrel{\widehat=} \langle P, \langle Q, \langle R, \langle S, T, \langle U, \langle V, W, X \rangle, Y \rangle\rangle\rangle\rangle\rangle$$

This representation is used in the abstract syntax below.

$$\begin{aligned} Z\_Schema\_Composition &== \mathrm{seq}_1\, Z\_Schema\_Hiding \\ Z\_Schema\_Conjunction &== \mathrm{seq}_1\, Z\_Schema\_Composition \\ Z\_Schema\_Disjunction &== \mathrm{seq}_1\, Z\_Schema\_Conjunction \\ Z\_Schema\_Implication &== \mathrm{seq}_1\, Z\_Schema\_Disjunction \\ Z\_Schema\_Equivalence &== \mathrm{seq}_1\, Z\_Schema\_Implication \end{aligned}$$

$$Z\_Schema\_Quantifier ::= Schema\_Universal \mid Schema\_Existential$$

$$\begin{aligned} Z\_Schema\_Exp \; ::= \; & Quantified\_Schema \langle\!\langle Z\_Schema\_Quantifier \times \\ & \quad Constrained\_Vars \times Z\_Schema\_Exp \rangle\!\rangle \\ \mid \; & Unquantified\_Schema \langle\!\langle Z\_Schema\_Equivalence \rangle\!\rangle \end{aligned}$$

$$\begin{aligned} Z\_Schema\_Body ::= \; & Text\_Form \langle\!\langle Constrained\_Vars \rangle\!\rangle \\ \mid \; & Expression\_Form \langle\!\langle Z\_Schema\_Exp \rangle\!\rangle \end{aligned}$$

Symbol table

The VDM-SL definition of a symbol table given in Chapter 10 is now summarized in a Z translation. The symbol table is used in the expansion of schema expressions to single schemas.

The symbol table is simply a partial function from identifiers to bodies of schemas.

$$Symbol\_Table == Identifier \mathrel{+\!\!\!\!\rightarrow} Z\_Schema\_Body$$

The translator has a single instance of this symbol table which will be referenced by the schema *Translator_Symbol_Table*.

Translator_Symbol_Table
$st: Symbol\_Table$

The operations to add elements to the table and update the table are specified in Chapter 10.

Expanding the body of a schema

The schema bodies associated with a name in the symbol table are possibly schema expressions requiring recursive application of the expression expanding function. For example, given the following Z schema definitions, the expansion of T requires the expansion of R. This is achieved by a recursive invocation of the function.

$$P \mathrel{\widehat=} [\ \ldots \mid \ldots\ ]$$

$$Q \mathrel{\widehat=} [\ \ldots \mid \ldots\ ]$$

$$R \mathrel{\widehat=} P \land Q$$

$$S \mathrel{\widehat=} [\ \ldots \mid \ldots\ ]$$

$$T \mathrel{\widehat=} R \land S$$

The expansion of a schema expression is defined in (Spivey, 1988a) and is not repeated here.

$$expand\_exp : Symbol\_Table \times Z\_Schema\_Exp \to Constrained\_Vars$$

$$\ldots$$

$$expand\_body : Symbol\_Table \times Z\_Schema\_Body \to Constrained\_Vars$$

$$\begin{array}{l} \forall\, st : Symbol\_Table;\ vars : Constrained\_Vars;\ exp : Z\_Schema\_Exp \cdot \\ \quad expand\_body\ (st, Text\_Form\ (vars)) = vars \land \\ \quad expand\_body\ (st, Expression\_Form\ (exp)) = expand\_exp\ (st, exp) \end{array}$$

The *Expand_Schema_Body* operation has an unusual signature. In addition to requiring the schema body as an input and the translator symbol table for reference, state-before and state-after schema texts are declared. The state-before text is not of interest but must be included for type correctness of the signature. This illustrates a minor notational inconvenience of Z; when describing state-based operations, a pair of variables must be introduced whether both are used or not. Use of the Δ convention for introducing the pairs of variables eases this difficulty.

The state-after text is necessary here in order to be able to compose this operation sequentially with others in order to define more complex operations. When the schema composition operator, '$\fatsemi$', is used a state-after variable must appear in the signature of the left schema operand.

The body of the schema is expanded using the *expand_body* function and the translator symbol table.

Expand_Schema_Body

$$\begin{array}{l} schema\_body? : Z\_Schema\_Body \\ schema\_text, schema\_text' : Constrained\_Vars \\ Translator\_Symbol\_Table \\ \hline schema\_text' = expand\_body\ (st, schema\_body?) \end{array}$$

Datatypes and operations

It is now possible to specify the abstract syntax of named parameterised datatypes and operations in VDM-SL and schemas in Z. One feature to notice is that a VDM-SL operation body has two predicates, corresponding to a pre-condition and a post-condition.

The *VDM_Data_Or_Op_Body* type has two constructors—the *VDM_Datatype_Body* function and the *VDM_Operation_Body* function. The *VDM_Data_Or_Op* type also has two constructor functions. The *Z_Schema_Defn* type has only one.

$$\begin{aligned} \mathit{VDM\_Data\_Or\_Op\_Body} ::= {} & \mathit{VDM\_Datatype\_Body}\langle\langle \mathit{Constrained\_Vars}\rangle\rangle \\ \mid {} & \mathit{VDM\_Operation\_Body}\langle\langle \mathrm{iseq}_1\, \mathit{Declaration} \times \\ & \quad \mathrm{seq}\, \mathit{Predicate} \times \mathrm{seq}\, \mathit{Predicate}\rangle\rangle \end{aligned}$$

$$\begin{aligned} \mathit{VDM\_Data\_Or\_Op} ::= {} & \mathit{VDM\_Datatype}\langle\langle \mathit{Identifier} \times \mathrm{seq}\, \mathit{Identifier} \times \\ & \quad \mathit{VDM\_Data\_Or\_Op\_Body}\rangle\rangle \\ \mid {} & \mathit{VDM\_Operation}\langle\langle \mathit{Identifier} \times \mathrm{seq}\, \mathit{Identifier} \times \\ & \quad \mathit{VDM\_Data\_Or\_Op\_Body}\rangle\rangle \end{aligned}$$

$$\mathit{Z\_Schema\_Defn} ::= \mathit{Schema\_Defn}\langle\langle \mathit{Identifier} \times \mathrm{seq}\, \mathit{Identifier} \times \mathit{Z\_Schema\_Body}\rangle\rangle$$

The translation of a schema text to a VDM-SL datatype is easily achieved because constrained variables are used to represent both schema bodies and VDM-SL datatype bodies.

Note that this operation has an output schema body but no corresponding input, i.e. no *schema_body*?. The initial value of the schema text is accepted from the state before the operation.

Translate_Schema_Text_As_Datatype

$$\begin{array}{|l} \mathit{schema\_text}: \mathit{Constrained\_Vars} \\ \mathit{schema\_body}!: \mathit{VDM\_Data\_Or\_Op\_Body} \\ \hline \mathit{schema\_body}! = \mathit{VDM\_Datatype\_Body}\ (\mathit{schema\_text}) \end{array}$$

This operation is composed with the *Expand_Schema_Body* operation defined earlier to give the specification of an operation which translates schema bodies to VDM-SL datatype bodies by first expanding them to schema texts and then translating the schema texts.

$$\begin{aligned} & \mathit{Translate\_Schema\_Body\_As\_Datatype} \mathrel{\widehat=} \\ & \qquad \mathit{Expand\_Schema\_Body} \mathbin{⨟} \mathit{Translate\_Schema\_Text\_As\_Datatype} \end{aligned}$$

It is now possible to construct the set of schema definitions which represent datatypes in VDM-SL terms. These are simply the schema definitions which do not include decorated variables—i.e. variables whose names end with ?, ! or ′.

This set is constructed in several steps. As an aid to remembering the purpose of these sets, all are given a zero subscript. The set of undecorated identifiers is called Id_0.

$$Id_0 == \{\ id: \mathrm{seq}_1\, \mathit{CHAR};\ s: \mathit{Script} \cdot \mathit{Id}\ (\mathit{Unspecified}, id, s, \mathit{State\_Before})\ \}$$

Now define the set of declarations which consist only of undecorated variables. Since there are two types of declarations the appropriate subsets are constructed and then united.

The set definitions below use the more unusual $\{\ vars \cdot exp\ \}$ construction of Z. As described on page 122, this form complements the more usual $\{\ vars \mid pred\ \}$ form. The choice between the two forms is made in order to increase the clarity of the description of the set.

$$Structured\_Variables_0 == \{\ id\colon Id_0 \cdot Constrained\_Variables\ (id)\ \}$$

$$Variables_0 == \{\ ids\colon \mathrm{seq}\, Id_0;\ type\colon Expression \cdot Variables\ (ids, type)\ \}$$

$$Declaration_0 == Structured\_Variables_0 \cup Variables_0$$

Now define the set of constrained variables whose declarations do not contain decorated variables. Using this, define schema bodies consisting of schema texts which are undecorated constrained variables. Using this, construct the set of named parameterised schemas whose variables are undecorated.

$$\begin{aligned}&Constrained\_Vars_0 == \\ &\quad \{\ vars\colon Constrained\_Vars \mid decs\ (vars) \in \mathrm{iseq}_1\, Declaration_0\ \}\end{aligned}$$

$$\begin{aligned}&Z\_Schema\_Body_0 == \\ &\quad \{\ vars\colon Constrained\_Vars_0 \cdot Text\_Form\ (vars)\ \}\end{aligned}$$

$$\begin{aligned}&Named\_Datatype_0 == \\ &\quad \{\ id\colon Identifier;\ s\colon \mathrm{seq}\, Identifier;\ body\colon Z\_Schema\_Body_0 \cdot \\ &\qquad Schema\_Defn\ (id, s, body)\ \}\end{aligned}$$

Translating schemas as datatypes

Having invested time in defining the set of schemas which represent datatypes, it is now only necessary to test for membership of this set in order to decide whether to translate the schema as a datatype or as an operation.

$$\begin{array}{|l}
\hline
\textit{Translate_Schema_As_Datatype} \\
schema_defn?\colon Z_Schema_Defn \\
schema_defn!\colon VDM_Data_Or_Op \\
\hline
schema_defn? \in Named_Datatype_0 \\
\exists\, s\colon \mathrm{seq}\, Identifier;\ Translate_Schema_Body_As_Datatype \cdot \\
\quad schema_defn? = Schema_Defn\ (head\ (s), tail\ (s), schema_body?) \land \\
\quad schema_defn! = VDM_Datatype\ (head\ (s), tail\ (s), schema_body!) \\
\hline
\end{array}$$

The calculation of pre- and post-conditions

The pre- and post-conditions calculated for the VDM-SL equivalents of the Z schemas are not the pre- and post-conditions which are defined in (Hayes, 1987). There, the conditions which are calculated are the *implicit* pre- and post-conditions. These are both predicates over single states, obtained by existentially quantifying by the output and state-after variables over the schema predicate to obtain the pre-condition and by the input and state-before variables to obtain the post-condition.

The post-condition thus generated is inappropriate because VDM post-conditions are predicates over *two* states. The pre-condition is inappropriate because it is vacuous. The VDM equivalent is non-trivial and the Z-to-VDM-SL translator generates a non-trivial pre-condition. Consequently, a user of the translator will be required to undertake proofs that the generated pre-condition is strong enough to ensure that the post-condition is attainable for all possible starting states. Fortunately, this proof obligation is exactly Jones' implementability proof obligation for operations (Jones, 1990). A disciplined Z author will already have discharged all of the implementability proof obligations.

The pre- and post-conditions which are generated have the advantage of being simpler to calculate than the traditional Z equivalents. Partition the schema predicate into a sequence which contains only those conjuncts of the predicate whose free variables are the state-before and input variables of the operation and a sequence containing the remaining conjuncts. The former is the pre-condition; the latter is the post-condition.

Define a set, *Input_Vars*, which contains only input variables, i.e. those variables whose mode has been designated as *Input*.

$$Input\_Vars == \{\ id\colon \mathrm{seq}_1\ CHAR;\ s\colon Script \cdot Id\ (Unspecified, id, s, Input)\ \}$$

Now define a set, $Pred_0$, of predicates whose free variables are state-before or input variables. The definition given below assumes the existence of a function *freevars* which returns the set of free variables of a predicate. A definition is not given for this function because it is sufficiently well understood.

$$Pred_0 == \{\ p\colon Predicate \mid freevars\ (p) \subseteq\ (Id_0 \cup Input\_Vars)\ \}$$

Now restrict the predicate sequence of the schema using this set and the familiar range restriction and subtraction operators in order to yield functions from the natural numbers to predicates. The *squash* function defined in (Hayes, 1987) is used below to 'compress' the functions to yield sequences.

A generic definition of the *squash* function is given in (Hayes, 1987). The function converts arbitrary functions with domain $\mathbb{N}_1$ into sequences by squashing the domain of the function e.g.

$$squash\ (\{\ 2 \mapsto A, 27 \mapsto C, 4 \mapsto B\ \}) = \langle A, B, C \rangle.$$

The following functions define the pre- and post-conditions to be the sequences which result from selecting the relevant predicates from a sequence of predicates.

$$
\begin{array}{|l}
pre, post\colon \mathrm{seq}\ Predicate \to \mathrm{seq}\ Predicate \\
\hline
\forall s\colon \mathrm{seq}\ Predicate \cdot \\
\qquad pre\ (s) = squash\ (s \rhd Pred_0) \land \\
\qquad post\ (s) = squash\ (s \mathbin{\rhd\!\!\!-} Pred_0)
\end{array}
$$

This completes the specification of the calculation of pre- and post-conditions.

Translating schemas as operations

In this section the translation of schemas as VDM operations is specified. Given a schema text in the form of constrained variables, the declarations are copied and the *pre* and *post* functions are applied to the predicate.

It is customary for the declarations associated with a VDM operation to be partitioned into sets of input variables, result variables and state variables. This task is performed by the VDM-SL unparser.

$$\begin{array}{|l}
\quad Translate_Schema_Text_As_Operation \\
\hline
schema_text: Constrained_Vars \\
schema_body!: VDM_Data_Or_Op_Body \\
\hline
\exists\, d: \mathrm{iseq}_1\, Declaration;\ p: \mathrm{seq}\, Predicate \cdot \\
\qquad schema_text = Constrained\ (d, p) \land \\
\qquad schema_body! = VDM_Operation_Body\ (d, pre\ (p), post\ (p)) \\
\hline
\end{array}$$

In order to make this operation more useful, it is composed with the operation which expands a schema body to constrained variables.

$$\begin{array}{l}
Translate_Schema_Body_As_Operation \mathrel{\widehat{=}} \\
\qquad Expand_Schema_Body \mathbin{⨾} Translate_Schema_Text_As_Operation
\end{array}$$

Now schema definitions may easily be translated as operations after first testing that the schema does not represent a datatype.

$$\begin{array}{|l}
\quad Translate_Schema_As_Operation \\
\hline
schema_defn?: Z_Schema_Defn \\
schema_defn!: VDM_Data_Or_Op \\
\hline
schema_defn? \notin Named_Datatype_0 \\
\\
\exists\, s: \mathrm{seq}\, Identifier;\ Translate_Schema_Body_As_Operation \cdot \\
\qquad schema_defn? = Schema_Defn\ (head\ (s), tail\ (s), schema_body?) \land \\
\qquad schema_defn! = VDM_Operation\ (head\ (s), tail\ (s), schema_body!) \\
\hline
\end{array}$$

This done, the two operations which translate schema definitions are disjoined to form a single operation which will translate any schema definition.

$$\begin{array}{l}
Translate_Schema_Defn \mathrel{\widehat{=}} \\
\qquad Translate_Schema_As_Datatype \lor Translate_Schema_As_Operation
\end{array}$$

This completes the specification of the treatment of schemas by the Z-to-VDM-SL translator.

Axiomatic definitions and mappings

In Z, function and constant values with global scope are defined using axiomatic definitions. In VDM-SL, mappings are used for the same purpose. Although the two definitions have very different representations in the concrete syntaxes of the two languages, they share the same abstract syntax form: constrained variables with a sequence of generic parameters.

Aside: In fact, several mappings would be needed to translate a single axiomatic definition if the Z table builder module had not unwound the axiomatic definitions to ensure that each introduced exactly one function or constant.

$$Z\_Axiomatic\_Defn ::= Axiomatic\_Defn \langle\!\langle Constrained\_Vars \times \mathrm{iseq}\, Identifier \rangle\!\rangle$$
$$VDM\_Mapping ::= Mapping \langle\!\langle Constrained\_Vars \times \mathrm{iseq}\, Identifier \rangle\!\rangle$$

Because the two structures share the same abstract syntax, translation is easily specified by noting that there are constrained variables and a sequence of identifiers which may be used to form the input and output using the relevant constructor functions.

$$\begin{array}{l}
\textit{Translate_Axiomatic_Defn} \\
\hline
ax_defn?: Z_Axiomatic_Defn \\
ax_defn!: VDM_Mapping \\
\hline
\exists\, vars: Constrained_Vars;\ ids: \mathrm{iseq}\, Identifier \cdot \\
\quad ax_defn? = Axiomatic_Defn\ (vars, ids) \land \\
\quad ax_defn! = Mapping\ (vars, ids) \\
\hline
\end{array}$$

Definitions

Definitions in Z are either schema definitions or axiomatic definitions. Definitions in VDM-SL are either datatype, operation or mapping definitions. The first two are grouped into a category of datatypes or operations in order to have the VDM-SL abstract syntax resemble the Z syntax as much as possible.

$$\begin{array}{lll}
Z_Definition & ::= & A_Schema_Defn \langle\!\langle Z_Schema_Defn \rangle\!\rangle \\
 & \mid & An_Axiomatic_Defn \langle\!\langle Z_Axiomatic_Defn \rangle\!\rangle \\
VDM_Definition & ::= & Datatype_Or_Operation \langle\!\langle VDM_Data_Or_Op \rangle\!\rangle \\
 & \mid & A_Mapping \langle\!\langle VDM_Mapping \rangle\!\rangle
\end{array}$$

Using the *Translate_Schema_Defn* and *Translate_Axiomatic_Defn* schemas defined earlier, the translation of schema definitions and axiomatic definitions is easily achieved by applying the definition constructor functions in reverse.

$$\begin{array}{l}
\textit{Translate_Schema_Definition} \\
\hline
defn?: Z_Definition \\
defn!: VDM_Definition \\
\hline
\exists\, Translate_Schema_Defn \cdot \\
\quad defn? = A_Schema_Defn\ (schema_defn?) \land \\
\quad defn! = Datatype_Or_Operation\ (schema_defn!) \\
\hline
\end{array}$$

$$\begin{array}{l}
\textit{Translate_Axiomatic_Definition} \\
\hline
defn?: Z_Definition \\
defn!: VDM_Definition \\
\hline
\exists\, Translate_Axiomatic_Defn \cdot \\
\quad defn? = An_Axiomatic_Defn\ (ax_defn?) \land \\
\quad defn! = A_Mapping\ (ax_defn!) \\
\hline
\end{array}$$

Now the translation of definitions is described by disjoining the above schemas.

$$\begin{array}{l} \mathit{Translate\_Z\_Definition} \mathrel{\widehat=} \\ \quad \mathit{Translate\_Schema\_Definition} \lor \mathit{Translate\_Axiomatic\_Definition} \end{array}$$

This completes the specification of the treatment of definitions by the Z-to-VDM-SL translator.

Phrases

A single Z phrase is either a Z definition—as explained above—or a list of the sets the specification writer takes as 'given', i.e. they are assumed to exist but nothing else is assumed about them. The set of characters, $CHAR$, is the only given set used in this specification.

$$\begin{array}{lll} \mathit{Z\_Phrase} & ::= & \mathit{Definition}\langle\!\langle \mathit{Z\_Definition} \rangle\!\rangle \\ & | & \mathit{Given\_Set\_Def}\langle\!\langle \mathrm{iseq}\, \mathit{Identifier} \rangle\!\rangle \\ \mathit{VDM\_Phrase} & ::= & \mathit{A\_VDM\_Definition}\langle\!\langle \mathit{VDM\_Definition} \rangle\!\rangle \\ & | & \mathit{VDM\_Assumed\_Sets}\langle\!\langle \mathrm{iseq}\, \mathit{Identifier} \rangle\!\rangle \end{array}$$

A schema is devoted to each category. The $\mathit{Translate\_Z\_Definition}$ schema is used to translate Z definitions.

$$\begin{array}{|l} \hline \mathit{Translate\_Definition} \\ \mathit{phrase?}: \mathit{Z\_Phrase} \\ \mathit{phrase!}: \mathit{VDM\_Phrase} \\ \hline \exists\, \mathit{Translate\_Z\_Definition} \cdot \\ \quad \mathit{phrase?} = \mathit{Definition}\ (\mathit{defn?}) \land \\ \quad \mathit{phrase!} = \mathit{A\_VDM\_Definition}\ (\mathit{defn!}) \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \mathit{Translate\_Given\_Sets} \\ \mathit{phrase?}: \mathit{Z\_Phrase} \\ \mathit{phrase!}: \mathit{VDM\_Phrase} \\ \hline \exists\, s: \mathrm{iseq}\, \mathit{Identifier} \cdot \\ \quad \mathit{phrase?} = \mathit{Given\_Set\_Def}\ (s) \land \\ \quad \mathit{phrase!} = \mathit{VDM\_Assumed\_Sets}\ (s) \\ \hline \end{array}$$

In order to translate a phrase, these schemas are disjoined.

$$\mathit{Translate\_Phrase} \mathrel{\widehat=} \mathit{Translate\_Definition} \lor \mathit{Translate\_Given\_Sets}$$

Texts

The final abstract syntax classes introduced are $\mathit{Z\_Text}$ and $\mathit{VDM\_Text}$. These are either phrases, as introduced above, or explanatory natural language text, or an end of document marker. Explanatory text is accompanied by a document position giving the row and column position of the text in the document. This is used as formatting information by the VDM-SL unparser.

$$
\begin{aligned}
&\mathit{Document\_Position} == \mathbb{N} \times \mathbb{N}\\[4pt]
&\mathit{Z\_Text} ::= \mathit{End\_Of\_Document}\\
&\quad\mid \mathit{Explan\_Text} \langle\!\langle \mathrm{seq}_1\ \mathit{CHAR} \times \mathit{Document\_Position} \rangle\!\rangle\\
&\quad\mid \mathit{A\_Z\_Phrase} \langle\!\langle \mathit{Z\_Phrase} \rangle\!\rangle\\[4pt]
&\mathit{VDM\_Text} ::= \mathit{VDM\_End\_Of\_Document}\\
&\quad\mid \mathit{VDM\_Explan\_Text} \langle\!\langle \mathrm{seq}_1\ \mathit{CHAR} \times \mathit{Document\_Position} \rangle\!\rangle\\
&\quad\mid \mathit{A\_VDM\_Phrase} \langle\!\langle \mathit{VDM\_Phrase} \rangle\!\rangle
\end{aligned}
$$

The translation of the end of document marker is trivial; explanatory text is passed on without translation; the *Translate_Phrase* schema is used to translate a Z phrase.

$$
\begin{array}{|l}
\hline
\mathit{Translate\_End\_Of\_Document}\\
\mathit{text?}: \mathit{Z\_Text}\\
\mathit{text!}: \mathit{VDM\_Text}\\
\hline
\mathit{text?} = \mathit{End\_Of\_Document}\\
\mathit{text!} = \mathit{VDM\_End\_Of\_Document}\\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{Translate\_Explan\_Text}\\
\mathit{text?}: \mathit{Z\_Text}\\
\mathit{text!}: \mathit{VDM\_Text}\\
\hline
\exists\, s: \mathrm{seq}_1\ \mathit{CHAR};\ p: \mathit{Document\_Position} \cdot\\
\qquad \mathit{text?} = \mathit{Explan\_Text}\ (s, p) \land\\
\qquad \mathit{text!} = \mathit{VDM\_Explan\_Text}\ (s, p)\\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{Translate\_Z\_Phrase}\\
\mathit{text?}: \mathit{Z\_Text}\\
\mathit{text!}: \mathit{VDM\_Text}\\
\hline
\exists\, \mathit{Translate\_Phrase} \cdot\\
\qquad \mathit{text?} = \mathit{A\_Z\_Phrase}\ (\mathit{phrase?}) \land\\
\qquad \mathit{text!} = \mathit{A\_VDM\_Phrase}\ (\mathit{phrase!})\\
\hline
\end{array}
$$

The schemas are now disjoined to give a schema which translates Z text to VDM-SL text.

$$
\begin{aligned}
&\mathit{Translate\_Text} \mathrel{\widehat=}\\
&\qquad \mathit{Translate\_End\_Of\_Document} \lor\\
&\qquad \mathit{Translate\_Explan\_Text} \lor\\
&\qquad \mathit{Translate\_Z\_Phrase}
\end{aligned}
$$

This completes the specification of the Z-to-VDM-SL translator.

12.2 Implementation

This specification differs slightly from the previous specifications in that a larger number of datatypes are defined here than are defined in any of the others. Implementations of these datatypes must be provided. Fortunately, many of the datatypes have obvious implementations in Ada. The sequences are readily represented by linked lists: the composite objects are readily implemented using the record construct. Semantic information—such as the injective nature of certain sequences—cannot be represented in the Ada types and must be reserved for the procedures and functions of the package.

No proof of satisfaction was produced for the implementation.

12.3 Conclusions

12.3.1 Learning specification languages

In learning a second specification language, the difficulty is not in mastering the library of standard datatypes and functions. Datatypes such as sets and sequences and the associated operations are common to many specification languages. Understanding the datatype building and operation structuring facilities is more taxing.

This difference is exemplified when translating between such languages. The majority of the work is needed to re-shape the operations and datatypes. The standard functions and datatypes may be appropriated or defined generically in the target language.

Part III

Analysis & Conclusions

Chapter 13

Analysis

In Part 2 of this thesis the specifications and development to implementation of the modules of a specification translator are presented. The novelty in this correctness-oriented development lies in the fact that components of the implementation are tested by proof and never by execution. In conventional practice, execution is more often favoured as the test mechanism; it is rare to find sketch proofs being submitted instead of test results.

The completion of the implementation of the translator affords the author the opportunity to assess the success or failure of the development and to substantiate or refute the implementation proofs by testing the implementation. Several aspects of the implementation are analysed:

1. the correctness of the development;
2. the clarity of the implementation;
3. ease of error diagnosis;
4. ease of error repair; and
5. program efficiency.

At this point, it is worthwhile to recall EW Dijkstra's aphorism about testing non-trivial computer programs.

> Testing can reveal the presence of bugs, never their absence.

The author acknowledges the fundamental limitation of testing embodied in Dijkstra's remark. The translator program was developed with proofs which are of varying degrees of rigour. It is not suggested that testing by execution will lead to the elimination of any remaining errors from the program: only that it *may* reveal errors which were not detected during construction of a *non-formal* correctness proof.

The potential of testing for the detection of errors which were not disclosed by proof does not mean that testing is a more exacting process than proof construction, only that the two activities are very different. Together, proof and testing may succeed in detecting an implementation error where (sketch or rigorous) proof construction alone has failed. If a fully formal proof had been constructed for every component of the implementation then testing by execution would be unnecessary. The reader is reminded that some parts are defended only by sketch proofs and that others have no supporting proof text.

It is significant to note *where* the errors occur in the translator and to observe an underlying pattern in these errors. This provides the insight into the correctness-oriented development of software systems which is the most significant contribution of this thesis.

The first section of this chapter presents the approach adopted by the author to the problems of error detection and correction. The working environment is reviewed in the second section.

The next section discusses the modules of the translator in the order in which they were presented in Part 2 of this thesis. The errors found in each are expounded and—where possible—likely causes of the errors are suggested. The order of presentation bears no relationship to the order in which the errors were identified by the author. The order of discovery is not thought to be significant—it is only a consequence of the test data sets chosen and the placement of the modules in the program superstructure. The order of discovery is not recorded here.

In some cases the specification is found to be inadequate because it fails to capture the informal requirement for that component of the translator. These errors are also appraised.

The concluding section of this chapter presents the core of the analysis of the development of the translator and communicates the observed pattern in the errors.

13.1 Error detection and correction

The detection of logical errors in the implementation and its supporting proof text is achieved by applying the executable image to sample test specifications. The sample test specifications are created by editing a legal *f*UZZ input to produce a type-checked Z specification in the concrete syntax of (King *et al.*, 1988). Care must be taken not to use those features of the *f*UZZ input language which are not supported by the input language of the Z-to-VDM-SL translator.

In some cases a component of the translator will fail with a run-time error. The Ada term for an error which occurs during program execution is an *exception.* The exception handling facility of Ada is not widely used in the implementation of the Z-to-VDM-SL translator and exceptions will always terminate the program. The program will rarely be terminated 'cleanly'. Only exceptions detected by the source handler will be handled by the program itself. These exceptional cases are listed in the development of the source handler in Chapter 6 on page 40. More often, one task will fail and the remaining tasks will deadlock.

The testing strategy applied to the program is so-called 'big bang' integration. In essence, this is system testing applied to the entire implementation without any prior 'unit' or 'isolation' testing of the modules of the system. The adoption of this approach consigned some of the modules to a delay of over three years between completion and first execution. All of the modules of the translator were re-compiled upon completion of the entire implementation to avoid possible difficulties caused by the development and subsequent release of revisions of the Ada compiler used.

13.2 The working environment

Several conventional software development tools were used in the creation and rectification of the Z-to-VDM-SL translator. The use of such tools is not incompatible with the use of correctness-oriented approaches to software development although it is noted that the tools discussed in this section are not integrated with the type-checkers discussed earlier. The same tools were used in the development of all of the modules of the translator in order to maintain fairness in the present experimental development.

13.2.1 Preparation of machine-readable texts

The author prepared the machine-readable text of the implementation on a VAX 8650 running VMS. A language-sensitive editor for Ada was available (VAX LSE). This provided syntax-directed editing via templates for the data structures, control structures and subprograms of the language. The editor is also integrated with the VAX Ada compiler to allow rapid location of compilation errors within the file.

The process of preparing a machine-readable text is accelerated by such tools because the use of editing templates reduces the initial number of syntax errors. The integration of the editor with the compiler allows simple syntax errors—such as omitted semicolons at the end of a statement—to be easily located within the text and corrected.

13.2.2 Compilation

An Ada compilation support system (ACS) was used to administer a library of copies of source text and object files. It was entrusted with the duty of ensuring that the separately compiled texts are mutually consistent. Consistency is achieved through comparing the times of the latest compilation of dependent modules.

The VAX Ada compiler offers a number of compilation options. One of these is the ability to have the compiler optimize the object code to increase execution speed or to decrease image size. This feature was not used. The author considers this type of optimization to be insecure. The construction of an Ada compiler is a difficult undertaking and the optimization of user programs is the most taxing part of that endeavour. In consequence the optimizing facility should be regarded as potentially incorrect. The UK Department of Defence draft standard for safety critical software (DEF STAN 00-55) recommends that optimizing compilers should *not* be used in the development of safety critical software. It recommends the use of a high-level language with a validated compiler.

The VAX Ada compiler provides several extensions to the Ada language. These features were not used in the implementation of the Z-to-VDM-SL translator.

13.2.3 Execution

The ACS and LSE tools do not assist with the detection of logical errors. The tool provided to assist with this problem is the VAX Symbolic Debugger. This provides—among other features—step-by-step interpretation of programs, the ability to examine the values of variables and the ability to evaluate expressions in the current program state. Such a tool is found to be useful. If an Ada program terminates with a run-time error very little diagnostic information is returned to the software developer. The following is a transcript of the diagnostic information delivered by the VAX Ada run-time system for a failed execution of the Z-to-VDM-SL translator.

```
CONSTRAINT_ERROR
Exception raised prior to PC = 00012F39
Task with ID %TASK 14 of type UNPARSER has
    terminated due to unhandled exception

CONSTRAINT_ERROR
Exception raised prior to PC = 000062A9
Task with ID %TASK 7 of type PARSER has
    terminated due to unhandled exception
```

In this case, two error messages have been generated. The first indicates that the unparser has failed to unparse a section of the specification which had been parsed and translated successfully. The unparser task raises an exception and terminates. After this occurs, the parser of the translator fails to parse a later section of the specification. This task also terminates after raising an exception. The remaining tasks continue to execute until either the task's input buffer becomes empty or its output buffer becomes full.

Very little of the above diagnostic message may be related easily to the program text. The names `UNPARSER` and `PARSER` do uniquely identify tasks in the author's program. However, this is only due to the author choosing a unique identifier for each task. If the same identifier had been used for two of the tasks in the program then the first step in utilizing the message would be to determine which task had failed. Further, the details of *why* and *where* the task failed are not given in a form which can be used immediately by the developer. The point at which the error occurred is related only to an approximate value of the program counter rather than being given as a line number in the source text. Further, the generic classification of `CONSTRAINT_ERROR` is given to many classes of errors, for example, numeric overflow or an attempt to dereference a null pointer. This generality further obscures the cause of the exception.

The assimilation of many different classes of errors into one very general error class seems to have been an unwise decision on the part of the Ada language designers. A hierarchy of error classifications would have been more useful to the software developer.

13.3 Errors in the development

In this section the errors in the development which the author has been able to detect by testing via observed execution are catalogued. The reader is also reminded of the errors which were detected by proof in Chapter 7 on page 56 and page 58. The errors which were detected by testing via observed execution are reported with the corrections which the author has implemented. This gives an impression of the severity of the error and the cost of correction ignoring the time taken to diagnose the cause of the error. At the time of writing, the author knows of no remaining errors in the implementation. As noted above, this does not guarantee that errors do not remain.

This section is a complete record of all the errors detected in the modules of the translator.

The error handler The author was unable to detect errors in this module.

The source handler The author was unable to detect errors in this module.

The lexical token generator The author was unable to detect errors in this module.

The terminal generator An error was found in the terminal generator which was developed in Chapter 8. The application of the type transfer function required in the addition on page 78 was incorrectly parenthesized as in the assignment statement below.

$$
\begin{aligned}
&Terminal.Position.Column :=\\
&\quad Token.Position.Column + Document\_Coordinate(I - 1);
\end{aligned}
$$

The version which appears on page 78 is the corrected version which is repeated below.

$$
\begin{aligned}
&Terminal.Position.Column :=\\
&\quad Token.Position.Column + Document\_Coordinate(I) - 1;
\end{aligned}
$$

This error was caused by carelessness on the part of the author who mistakenly believed that the two expressions were equivalent. They differ when I has the value 1. The former fails when calculating the sub-expression $Document\_Coordinate(I - 1)$ because 0 is not an allowable value of the *Document_Coordinate* type.

The consequence of this error is to cause the translator to fail at the first occurrence of a primed variable in the input. Thus this error was easily detected and attributed to the above assignment statement.

The generic bounded buffer This module is taken from (Barnes, 1989). Although the version there is a satisfactory implementation of a bounded buffer data structure the author failed to realize that it is a non-terminating task. This caused the translator to fail to terminate upon completing the translation of a specification. A **terminate** alternative was added to the **select** statement in the buffer to allow the task to terminate when all the other tasks terminated. The specification of the buffer is unchanged.

The input buffer of the pre-parser The author was unable to detect errors in this module.

The pre-parser The pre-parser and the parser are described in Chapter 9. Errors were detected in both the specification and the implementation of this module.

Specification error: The specification of the *PROCESS-BASENAME* operation on page 84 of Chapter 9 was discovered to be erroneous. The operation required an initial prefix of the input sequence to be removed. The predicate which attempted to capture this property is given below.

$$input = b \curvearrowright \overleftarrow{input}$$

This equation does not capture the required property. Recall that in VDM-SL the hooked variables denote *state-before* values. Thus the above predicate requests the sequence b to used as the initial part of the input after the operation is completed. This initial prefix is followed by the original input. The predicate is corrected on page 84 as shown below.

$$\overleftarrow{input} = b \frown input$$

This error is attributable to an unfortunate pun between the concrete syntax of the input language of the QUB VDM type-checker (McParland, 1989) and the concrete syntax of Z. In the input language of the QUB VDM type-checker, hooks are represented by a prime after the name. This, of course, is the Z convention for denoting *state-after* variables. The author had just completed the Z specifications in Chapters 6–8 and was more familiar with the Z notation than with VDM-SL when he commenced the specification given in Chapter 9.

Implementation errors: The implementation of the pre-parser contained two elementary errors.

1. The Ξ and Δ symbols which are used to prefix schema names were not consumed by accepting another symbol from the input buffer. This caused a subsequent misparse of the Greek letter prefix as a schema identifier.
2. The pre-parser did not generate an "end of document" symbol. This caused the parser to deadlock when its input buffer became empty. This prohibited the parser from generating its "end of document" token and this effect propagated through the remaining tasks.

The parser input buffer The author was unable to detect errors in this module.

The parser Three errors were detected in the parser component of the translator. These errors are interesting because they are very different.

1. The first error is somewhat technical and arose because of the author's failure to uncover a subtle feature of the treatment of variant records and access types in Ada. The values of the discriminants of an object designated by an access value cannot be changed. For example, the type *Predicate* used in the translator is a composite object with a variant part. A predicate may be quantified: the variant part contains $\forall$, $\exists$ or $\exists_1$ and the declaration of the quantifying variables. Lists of predicates are dynamically created by the parser. The creation of these lists must be achieved carefully in order to prevent an attempt to alter the value of a discriminant of an object designated by an access value.

   The following program fragment would not allow the procedure *Predicate* to update the variable denoted by the variable expression $P.\mathsf{all}$ even if the variable is passed as an **in out** parameter of the procedure.

   $$P := \mathsf{new}\ Predicate\_Rec;$$
   $$Predicate(P.\mathsf{all});$$

   The following program fragment does allow the variable denoted by $P.\mathsf{all}$ to be assigned the value resulting from the invocation of the *Predicate* procedure. This is achieved by introducing an auxiliary variable, *Pred_Rec*, which is used as the actual parameter of the procedure invocation.

```
declare
    Pred_Rec: Predicate_Rec;
begin
    Predicate(Pred_Rec);
    P := new Predicate_Rec'(Pred_Rec);
end;
```

This error in the treatment of composite objects designated by an access value occurs at three distinct points in the implementation of the parser. All three occurrences were rectified by introducing an auxiliary variable as shown above.

2. The construction of the parser predates the publication of (King *et al.*, 1988). The author based the specification and implementation upon a draft of the monograph which was obtained from the authors. In this draft, the names *Start_Seq* and *Left_Seq* are both used for the symbol which is used to commence the enumeration of a sequence literal in Z. These names were used in the implementation of the Z-to-VDM-SL translator in order to maintain consistency with the draft of the monograph.

   Sadly, the terminal generator generates the *Start_Seq* symbol whereas the initial version of the parser attempted to accept the *Left_Seq* symbol. The parser was corrected in order to accommodate the synonym. The synonym is assumed to have been accidently introduced by the authors of the draft of (King *et al.*, 1988). It is resolved in the final version of the monograph by using the *Left_Seq* identifier throughout.

   The lack of formality in linking adjoining modules of the specification must also be held partly accountable for this error.

3. After the pre-parser was corrected to generate an "end of document" token, it was discovered that the parser did not communicate this token to the table builder.

The generic symbol table The author was unable to detect errors in this module.

The *less_than* function This function defines an ordering upon identifiers. It forms a significant part of the implementation of the definition table but is sufficiently general and reusable that it is isolated as a compilation unit which is used by several of the other modules of the Z-to-VDM-SL translator. The author was unable to find errors in this function.

The definition table The definition table is a relatively simple instantiation of the generic symbol table. An alternative implementation was used during the process of error detection and correction. This implementation of the symbol table provided an operation to allow the binary search trees in the module to be traversed and to have the contents of each node printed in the concrete syntax given in (King *et al.*, 1988). As noted above, the author was unable to detect errors in the implementation of the symbol table. The facility to print the contents of the table was useful in diagnosing the errors in the modules which add definitions to the table.

The input buffer of the table builder The author was unable to detect errors in this module.

The *equal* function This function defines equality for identifiers. It forms a significant part of the implementation of the table builder. As with the *less_than* function, it is sufficiently general and reusable that it is separately compiled and used by several of the other modules. The author was unable to find errors in this function.

The table builder Two errors were found in the implementation of the table builder. The first is an omission which resulted from an omission in the specification. The second is a fundamental error which is frequently made by novice programming students.

1. The specification of the table builder module omits the specification of a function which expands the Ξ schema prefix replacing it by its definition. The schema

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\ \hline
\Xi S \\
decs \\ \hline
pred \\ \hline
\end{array}$$

is expanded by the function to give the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\ \hline
\Delta S \\
decs \\ \hline
S' = S \land \\
pred \\ \hline
\end{array}$$

It might be said that this function could as easily be said to have been omitted from one of the other modules of the specification. However, this module is the most suitable location for the function. As might be deduced from the example above, the function is relatively simple to implement and does not add significantly to the complexity of the implementation of the table builder.

2. The second error of the translator is an instance of a simple error which is frequently made by novice programming students. The procedure invocation which obtains a new token from the input buffer was omitted from the iterative construct in the body of the task. The input procedure was only invoked once immediately before the execution of the driving `loop` construct in the task body. The invocation should have been the first statement *within* the `loop` construct. This resulted in the first token being repeatedly placed in the input buffer of the core of the translator. The tasks before the table builder became deadlocked when their output buffers became full. The tasks after the table builder repeatedly removed the copies of the initial token from their input buffers. The initial token was copied repeatedly to the output file by the unparser.

   It is noted that this error would have been rapidly detected by an attempt at verification.

The author cites such errors as the latter given above as evidence that he is developing the Z-to-VDM-SL translator program in a very different manner from the structured or modular development method that he would previously have employed. Correctness-oriented methods of software development have a static, analytical character which contrasts with the dynamic, operational nature of conventional software development methods. This use of an alternative development process and the associated re-learning required has allowed the author to rediscover for himself the problems which a novice programming student encounters.

The translator input buffer The author was unable to detect errors in this module.

The $pred_0$ function The $pred_0$ function is a utility function which is employed by the translator core in determining which of the sequence of predicates in a Z schema should be used in the pre-condition of the VDM translation and which should be used in the post-condition of the translation. The function is Boolean-valued and will return a *true* result if the predicate is to be included in the pre-condition and *false* otherwise. Because the function is lengthy, it is isolated as a separate compilation unit in order to avoid obscuring the text of the translator core.

A single clerical error was found in this function. The function bases its classification of predicates upon the class of identifiers used in the predicate. No type information is required because identifiers in Z are labelled as inputs, outputs or state variables. The variable x is a variable in the state before, x' is a variable in the state after, $y?$ is an input and $z!$ is an output. This is more than a notational convention intended to assist the reader of a Z specification, it is the only mechanism provided in Z to consign variables to one of these classes.

The function erroneously classified predicates involving only state-before and state-after variables as being suitable for inclusion in the pre-condition. Predicates which involved input variables were erroneously classified as being unsuitable for inclusion in the pre-condition.

Again, this error was relatively easy to detect. The only function which classifies predicates is the $pred_0$ function. Since the predicates were being classified incorrectly, this function was diagnosed as the source of the error.

This error should serve to remind the reader of the massive amount of clerical detail in the text of a non-trivial program and the sigil of digital systems that even the alteration of a single bit of the executable image can radically transform the behaviour of the system.

Declaration utilities This package was constructed as a detachable subcomponent of the translator core. It is a separately compiled package which provides a range of functions which process lists of declarations of typed variables. The functions are used by several of the packages discussed later. The author was unable to find errors in this module.

The *expand_body* function The *expand_body* function is specified on page 128 in Chapter 12. The function is sufficiently complex to be partially specified by reference to the semantics of Z given in (Spivey, 1988a). The implementation of the specification is sufficiently lengthy not to be embedded in the implementation of the translator core. Rather, it is isolated as a separate compilation unit.

The reference in the specification of the function to the semantics of Z as given in (Spivey, 1988a) was thought to be a convenient abbreviation mechanism but—although all of the information is is to be found in the reference—this has proved to be an unwise decision. In the implementation of the function, the recursive expansion of schemas referenced in the declaration part of a schema was not performed. This has been corrected.

This error encourages the author to be less careless in the construction of specifications for future projects. One of the most significant insights provided by the use of correctness-oriented approaches to software development is the recognition of the difficulty of constructing specifications of high quality. It was foolhardy of the author to attempt to abbreviate the specification in the way mentioned. The reader is advised to be wary of such references and citations both in the reader's own specifications and in the specifications of others.

The translator Two errors were detected in this module.

1. The operation which expands schema texts which represent operations in VDM-SL did not add the expanded schema texts into the symbol table for use in the expansion of later schemas. This prevented the translation of schemas defined by a schema expression which had a subcomponent which was itself defined by a schema expression.
2. As with the pre-parser and the parser, the translator core did not initially produce an "end of document" token.

The unparser input buffer The author was unable to detect errors in this module.

The unparser The majority of the errors which were found in the implementation of the translator were found in the unparser. An explanation of this is given later.

1. A simple error was found in the treatment of identifiers by the unparser. Z identifiers use underscores in a long identifier such as *Not_Found*. The VDM-SL notation uses hyphens as in *Not-Found*. The underscores were not changed to hyphens in the initial version of the implementation.
2. The function which ascertains whether or not a variable has been declared as a state-after variable did not search the imported schemas which occur in the declaration list of a schema text.
3. The bar symbol in $decs \mid pred$ was generated if the predicate list was empty. In this case it should have been omitted.
4. In Z, schemas which represent datatypes with an invariant may be included in a schema which represents an operation. The following is an example of this mechanism.

Celsius

$temp : \mathbb{Z}$

$temp \geqslant -273$

$$
\begin{array}{|l}
\hline
\textit{Heat} \\
\Delta \mathit{Celsius} \\
\hline
\mathit{temp}' > \mathit{temp} \\
\hline
\end{array}
$$

The VDM-SL translation which the Z-to-VDM-SL translator produces is given below.

$\mathit{Celsius}\text{-}\mathsf{type} :: \mathit{temp} : \mathbb{Z}$
where
$\mathit{inv}\text{-}\mathit{Celsius}\text{-}\mathsf{type}(\mathit{mk}\text{-}\mathit{Celsius}\text{-}\mathsf{type}(\mathit{temp})) \triangleq$
$\quad \mathit{temp} \geqslant -273$

$\mathit{Heat}\ ()$
$\mathsf{ext\ wr}\ \mathit{Celsius} : \mathit{Celsius}\text{-}\mathsf{type}$
$\mathsf{post}\ \ \mathsf{let}\ \mathit{Celsius} = \mathit{mk}\text{-}\mathit{Celsius}\text{-}\mathsf{type}(\mathit{temp})\ \mathsf{in}$
$\qquad \mathsf{let}\ \overleftarrow{\mathit{Celsius}} = \mathit{mk}\text{-}\mathit{Celsius}\text{-}\mathsf{type}(\overleftarrow{\mathit{temp}})\ \mathsf{in}$
$\qquad \mathit{temp} > \overleftarrow{\mathit{temp}}$

The VDM-SL translation requires two identifiers. One is for the datatype itself; the other is for the instance of a value of that type. The Z notation requires only one identifier for both. The new identifier for the type in the VDM-SL translation is generated by adding a type suffix to the identifier. These suffixes were not initially generated in the applications of constructor *mk*-functions and invariant *inv*-functions which appear in a composite object invariant part.

5. The function which tests schemas to determine whether or not they include references to external variables was not sufficiently rigorous in its selection criteria. Any schema which includes another schema in its declaration section was deemed to have made reference to external variables whereas the schema name must be prefixed by Ξ or Δ in order for this assumption to hold.

6. The operation which generates $\mathsf{let} \ldots \mathsf{in}$ clauses at the start of a pre- or post-condition did not remove the Greek letter prefix from a schema name before attempting to retrieve its definition from the definition table. Thus, the definition was not found and could not be used to construct the $\mathsf{let} \ldots \mathsf{in}$ clause.

7. The operation which generates $\mathsf{let} \ldots \mathsf{in}$ clauses generated 'hooked' versions of the identifiers in these clauses in the pre-conditions of an operation. The hook over an identifier is the VDM-SL notation for a state-before variable. Since the pre-condition is a predicate over a single state the reference to the state before the operation is meaningless. The function was generalized by adding a parameter to the operation to indicate whether or not the hooked identifiers should be generated.

8. Operations without a pre-condition were not unparsed correctly. A placeholder for pre-conditions was generated.

9. Tuples were not unparsed correctly. The composite type which is used to represent tuples is given below.

```
type Exp_10_Rec(Tag: Base_Expression_Type := An_Identifier) is
  record
    case Tag is
      when An_Identifier =>
        Id: Identifier_Rec;
        Instantiation_Exp_List: Seq_Of_Expression_Ptr;
      when A_Natural_Number =>
        Nat: Identifier_Rec;
      when A_Set =>
        Set_Val: Set_Exp_Rec;
      when A_Sequence =>
        Seq_Expressions: Seq_Of_Expression_Ptr;
      when A_Parenthesized_Exp | A_Tuple =>
        Exp_List: Seq_Of_Expression_Ptr;
    end case;
  end record;
```

With hindsight, this type definition has an obvious weakness. Sequence literals and tuples are both parenthesized sequences of expressions and the two cases should have been folded together. (The tuple and the parenthesized expression are treated together to make the task of parsing the language easier.)

Separating sequence literals and tuples introduces two fields of identical type. This allows the two fields to be confused. Such an error cannot be detected by static type checking. The error of confusing the two fields was introduced in implementing the unparser. This caused an exception to be raised when attempting to access a field of the record which was not accessible due to the value of the tag field of the record.

10. One subtlety of the VDM-SL notation was overlooked in unparsing identifiers denoted by function applications. Components of a composite object may be accessed by applying a projector function to a value of the host type. These function names are never hooked. The unparser initially generated $\overleftarrow{x}(c)$ for $x(c)$ in the post-condition of an operation.

11. Finally, the processing of natural language text was initially incorrect. The end of a line in the text is denoted by including an "end of line" character in the string which contains the prose text. These characters were initially written to the output file rather than being taken as markers to indicate that the current line was complete.

The translator shell A compilation shell is required to cite the components of the translator named above to produce the executable image. This extremely simple procedure was initially incorrectly constructed. Several of the modules to be cited were omitted. This caused the translator to be miscompiled. The error was corrected by citing the modules using Ada with clauses.

This completes the inventory of the errors which were discovered in the implementation of the Z-to-VDM-SL translator through proof and testing by execution.

13.4 Use of the specification translator

The use of the specification translator on sample specifications is discussed before concluding observations from the project are presented.

The translator has been applied to a suite of sample specifications. One of the specifications in the suite is a component of the specification of the translator itself. The component chosen was the initial specification of the source handler of the translator. The Z specification is given in Chapter 6: the output from the translator is given in Appendix C. The output is a correct, intelligible VDM-SL translation of the input.

13.4.1 Efficiency of the translator

Although the efficiency of execution of the translator was not the author's prime concern during its construction, he did wish to produce an efficient product. The source handler specification is three pages long and was translated in under six seconds. This is considered to be significantly less than the maximum tolerable processing time for a specification of that length. The reader is reminded that this execution time is achieved without use of the optimizing features of the VAX Ada compiler. Also, the author has not tuned the implementation by choosing optimal sizes for the buffers between tasks.

One of the implementation techniques used by the author to improve the clarity of the implementation and to assist with the problems of relating a component of the implementation to its specification is the use of recursion as the principal repetition mechanism of the implementation. As would be expected, the use of parameter passing in the implementation is increased in consequence. In some of the modules of the translator this style is widely used. The essence of the style is side-effect free procedural programming with very little use of assignment and heavy reliance on recursion and parameter passing allowing the author to approximate a pseudo-functional style of implementation. This style is purely functional in many of the functions which process lists.

Many of the recursive procedures and functions are 'tail' recursive. That is, there is no further processing to be performed after completion of the recursive invocation of the subprogram. This form of recursion has the favourable consequence that the activation record of the current invocation may be re-used for the recursive invocation thereby improving the efficiency of the implementation. Whether or not the author's compiler uses this economical implementation technique, he is pleased that his reliance upon recursive processing has not adversely effected the efficiency of the completed implementation. It was always the author's belief that replacing tail recursive definitions by equivalent iterative constructs was at best a poor attempt at optimization and at worst a potential source of error.

13.4.2 Utility of the translator

The question of the efficiency of the translator has been answered above. The question of the usefulness of the translator may now be addressed. The translator has proved its usefulness as a specification exercising tool by detecting an error in its own specification.

An error in the sub-operation *Establish_Position* in the specification of the source handler (Chapter 6, page 38) had remained undetected by several readers over a period of three years before being uncovered by consulting the output of the Z-to-VDM-SL translator.

The *Establish_Position* operation included a $\Xi State$ declaration which, when the operation was imported into other schemas, rendered the importing schema inconsistent by contradicting the definition of the state-after variables in the importing schema. The declaration was weakened to $\Delta State$ in order to correct this error.

The error was uncovered by inspection of the output of the Z-to-VDM-SL translator. The VDM-SL translation of the operation requested both read and write access to the same variable. The author initially believed that he had uncovered another error in the implementation of the translator but subsequently realized that the expansion of the schema inclusion had indeed been correctly performed and that it was the input which was erroneous. It is believed that this error would have remained undetected for considerably longer had the specification translator not been used.

13.5 Observations

13.5.1 Number of implementation errors

The first observation to be made is how *few* errors were made in the implementation of the Z-to-VDM-SL translator. Although several pages are required to list and explain the errors, the author believes that more errors would have been expected had the implementation been created without the use of proof and refinement to create components of the implementation. Some of the reasons for this belief are given below.

- *The Z-to-VDM-SL translator is the author's first Ada program.* The lack of familiarity with features of the language such as the type transfer mechanism and the use of access variables account for some of the errors above although more were expected than were found. The author believes that he would have been less cautious about the use of less familiar features of the language if he had not been so concerned about the difficulty of defending development steps by proof.

- *The tasking mechanism and other advanced features of the language were used.* Although the author wished to work within a reliable subset of the language, the representation of the rich data structures of the Z and VDM-SL languages compelled him to use dynamic and variant data structures in their representation. The generic mechanism of the language is also used to make components of the implementation more general thereby allowing the components and their accompanying specification and proof to be used by other software developers.

  The tasking facilities of the language were used in the implementation in order to examine the degree to which the difficulty of the development of the implementation lay in the sequential processing of the specification by the tasks rather than in their concurrent interaction. The difficulty in this development was found to be in the creation of the sequential components. The parallel composition of these components forms a simple pipe structure.

- *The Z-to-VDM-SL translator is a non-trivial program.* The text of the program is approximately 10,000 lines long. It was created over a period of three years with some significant pauses to learn a new specification notation or a new software development methodology.

In part, the correctness of the result is attributed to the author's use of a 'clean' imperative programming style. The pseudo-functional style used results in the entire 10,000 lines of the implementation containing less than 500 assignment statements. These include all occurrences of the variable initializations which are performed as variable declarations are elaborated.

Again, the reader is reminded that the author cannot claim that the implementation is error-free, only that he has been unable to discover further errors. The correction of the errors given above was sufficient to produce an implementation which executes efficiently and translates the sample test specifications correctly.

13.5.2 The detection of errors

Errors in the implementation were detected rapidly. The author attributes this to the very detailed understanding of the desired behaviour of the translator which he acquired through construction of a formal specification for the translator.

The discovery of the underlying cause of the errors (as distinct from the location of the exception which terminated the program's execution) was also achieved rapidly. The modularity of the development and the resulting separation of concerns is believed to have contributed to the increase in the clarity of the text which allows causal diagnosis of the errors to be achieved quickly.

No significant additional implementation was required when correcting the errors in the translator. Also, no significant sections of the program were so badly erroneous that the author believed that they should be discarded and a re-implementation of the section undertaken. This is thought to be due to the thorough scrutiny of the problem which is required by the construction of a formal specification.

13.5.3 The observed pattern in the errors

The above taxonomy of the errors in the implementation of the Z-to-VDM-SL translator uncovers a clear and consistent pattern in the location of the errors. The errors occur in the modules which have least supporting proof text. Modules such as the symbol table, which has a rigorous development, could not be made to fail in testing.

Further, weaknesses in the specification of a component often led to corresponding errors in the implementation. The unparser is an extreme example of this phenomenon. The specification of that component is not teased apart from the specification of the translator core and the majority of the errors of the development are, consequently, to be found in that component. This is evidence that creating a clear and complete specification of a software system in itself assists with the problem of constructing the implementation.

Another observation to be drawn from the above concerns the selection of a specification language or methodology for software development. Choice of specification language or refinement style had no perceptible effect on the correctness of the components of the present experimental development. The author believes that it is the use of mathematical analysis which produces the benefits in correctness-oriented software development rather than the details of the notation used to express specifications or proofs.

CB Jones in (Jones, 1980) voices his belief that the benefit of the construction of a correctness proof diminishes as the proof is strengthened from a sketch proof to a rigorous proof with detail after detail being supplied until the proof is fully formal. Evidence for this belief has never been presented before. That evidence and its subsequent analysis is the unique contribution which this thesis makes to the development of the exacting craft of software engineering.

Chapter 14

Conclusions

A specification translator has been developed using an experimental development style. The 'variables' of the experiment were:

1. the choice of specification notation used in the specification of a module of the translator; and

2. the degree of rigour used in development of a module of the translator.

The 'constants' of the experiment were:

1. only one implementation language was used throughout;

2. there was only one specification author; and

3. there was only one implementor.

Although the record of the development itself is of interest, two significant observations were made of the completed development.

1. The notation used to express specifications and proofs of correctness had no perceptible effect on the correctness of the implementation.

2. The construction of proofs of correctness made an obvious positive contribution to the correctness of the implementation.

14.1 Use of specification languages

Although the author was free to use any feature of the specification languages he wished to use, certain features were never needed. This is an interesting observation since formal specification languages are relatively small formal languages.

Two differences between Z and VDM-SL are now noted. The first is the observation made on page 19 in Chapter 4 that Z provides fewer constructs for the expression of algorithmic designs than VDM-SL. The second difference arises from Z's use of a single predicate in the description of operations where a ⟨pre-condition, post-condition⟩ pair is used in VDM-SL. It is well understood that it is the use of a single predicate which allows a schema calculus to be defined for Z. However, it is less widely appreciated that the use of a single predicate allows more elegant specifications of operations to be constructed.

Consider the following specification of an operation which computes the integer square root of a natural number. The operation is easily specified by requiring the square of the output of the operation to be the input value.

$$Sqrt \mathrel{\widehat{=}} [x?, x!: \mathbb{N} \mid x? = x! * x!]$$

Of course, this operation may only be applied to numbers which are perfect squares. That pre-condition is not stated explicitly in the Z specification of the operation; rather, the specification author is invited to calculate the pre-condition. The following pre-condition would be calculated.

$$\mathrm{pre}\ Sqrt \mathrel{\widehat{=}} [x?: \mathbb{N} \mid \exists\, x!: \mathbb{N} \cdot x? = x! * x!]$$

In VDM-SL, the author must give the pre-condition as part of the operation in order to satisfy the implementability proof rules given in (Jones, 1990). For this problem, there is no more elegant pre-condition to be found than the pre-condition which was calculated above. The VDM-SL equivalent of the specification is less elegant due to the enforced repetition of the post-condition within the pre-condition. The VDM-SL equivalent of the specification is given below.

$$\begin{array}{l} Sqrt\ (x\text{-}\mathsf{in}: \mathbb{N})\ x\text{-}\mathsf{out}: \mathbb{N} \\ \mathsf{pre}\ \exists\, x\text{-}\mathsf{out} \cdot x\text{-}\mathsf{in} = x\text{-}\mathsf{out} * x\text{-}\mathsf{out} \\ \mathsf{post}\ x\text{-}\mathsf{in} = x\text{-}\mathsf{out} * x\text{-}\mathsf{out} \end{array}$$

14.2 Use of the implementation language

Although the author was free to use any feature of the implementation languages he wished to use, certain features were never needed. The reader should not be surprised by this observation. Ada is a large, *wide-spectrum* language which was designed to address a large number of application areas. Although these features were eschewed by the author, they should not be regarded as encouraging software development which is of poor quality. Some of the features were included in the language in order to assist the developers of low-level control software for embedded systems. These features were not required by the author.

An unusual style of Ada use developed as the implementation progressed. The later modules of the program are expressed in a pseudo-functional style which is, in essence, side-effect free procedural programming.

14.3 Notations for refinement

Formal notations for expressing software designs are less plentiful than formal specification languages. This is perhaps due to the difficulty of motivating the development of such languages and the care which must be taken in explaining their intended use. The author believes that the use of such languages would be of benefit in software developments where the cost of failure was not sufficiently high to justify fully formal development.

The use of a design language would partly bridge the gap between specifications and implementations thus providing formal documentation of design decisions. These decisions could be supported by proofs of satisfaction which are not all fully formal.

Consider now a 'Design Z': a development from Z which is intended to be used to express algorithmic refinements of specifications. This language would clearly be related to the language which is the basis of Carroll Morgan's Refinement Calculus (Morgan, 1990) but would be more closely tied to the Z notation.

The Z specification language is founded on typed set theory. The use of sets enables the author of a specification to specify predicates without any need to introduce explicitly a domain of truth values into the specification. An author of a Z specification constructs a typed set of elements which have a common, desired property. Testing for membership of this set allows the required elements to be found.

Using sets has an advantage over the use of a Boolean domain because it encourages the reader to treat the underlying logic of Z as simple two-valued logic. This is reinforced by the well-understood mathematical idea that an element is either a member of a set or it is not. There is no concept of the outcome of the test being *undefined.* The use of Boolean-valued functions would cause the reader to consider non-termination of the function application or undefinedness of the result.

In using Z to document algorithmic designs for software systems, the specifier has very different concerns from the issues which arise when using Z to create initial, very abstract software specifications. In design, the ability to construct conditional expressions is considered a useful tool.

Consider the following specification of the function which determines the smallest power of 2 which is not less than the strictly positive number which is supplied as the argument of the function.

The following utility specification describes the function which computes the set of all positive powers of a strictly positive number, x. The function returns the least set which is generated by including x in the set and $x * y$ in the set if y is in the set. This is specified by returning the distributed intersection of all such sets.

$$
\begin{array}{|l}
powers : \mathbb{N}_1 \rightarrow \mathbb{P}\,\mathbb{N}_1 \\
\hline
\forall\, x, y : \mathbb{N}_1 \cdot \\
\qquad powers\ (x) = \bigcap \{\ s : \mathbb{P}\,\mathbb{N}_1 \mid (x \in s) \land (y \in s \Rightarrow x * y \in s)\ \}
\end{array}
$$

Computing the least power of 2 not less than a given number may now be easily specified using the *min* function from the Z Basic Library.

$$
\begin{array}{|l}
nextpower : \mathbb{N}_1 \rightarrow \mathbb{N}_1 \\
\hline
\forall\, n : \mathbb{N}_1 \cdot \\
\qquad nextpower\ (n) = min\ \{\ x : \mathbb{N}_1 \mid x \in powers\ (2) \land x \geqslant n\ \}
\end{array}
$$

Now, a generic definition of a conditional expression mechanism is given.

$$\mathbb{B} ::= True \mid False$$

$$
\begin{array}{|l}
\hline\hline
[X] \\
cond : \mathbb{B} \times X \times X \rightarrow X \\
\hline
\forall\, x_1, x_2 : X \cdot \\
\qquad cond\ (True, x_1, x_2) = x_1 \land cond\ (False, x_1, x_2) = x_2 \\
\hline
\end{array}
$$

The expressivity and malleability of the Z language is illustrated here because the usual Z Basic Library may be replaced by a version for Design Z which includes the Boolean type defined here and redefines the relational operators to be functions which return Boolean results. In the next part of this development, assume that such a library has been constructed and that the *cond* function is included.

The refinement of the *nextpower* function uses an auxiliary function with an *accumulation parameter* to compute the powers of 2 until a sufficiently large result is obtained.

The auxiliary function is a conditional recursive function which may be expressed using the *cond* function defined above.

$$\begin{array}{|l}
nextpower_0 : \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{N}_1 \\
\hline
\forall\, m, n : \mathbb{N}_1 \cdot \\
\quad nextpower_0\ (m, n) = \\
\qquad cond\ (n \leqslant m, m, nextpower_0\ (2 * m, n))
\end{array}$$

$$\begin{array}{|l}
nextpower : \mathbb{N}_1 \rightarrow \mathbb{N}_1 \\
\hline
\forall\, n : \mathbb{N}_1 \cdot nextpower\ (n) = nextpower_0\ (2, n)
\end{array}$$

Finally, the conditional recursive function may be replaced by an iterative construct preceded by the appropriate variable initialization in a realization of the function in an imperative programming language although it is noted that the author does not consider this to be wholly necessary and would be content to employ a recursive function in a final, polished implementation.

$x := 2;$ while $x < n$ loop $x := x * 2;$ end loop;

14.4 Extensions and future work

14.4.1 Extensions

Changes to the input language

An obvious extension to the translator is the revision of the modules which process the input language to allow a different concrete syntax to be used. The input language of *f*UZZ is a suitable candidate. Some modest extensions to the translator would be required to allow it to process free types and injective sequences.

A Z with a notation for refinement

The ability to create subordinate symbol tables is provided by the generic symbol table which was developed for the Z-to-VDM-SL translator. This facility is not at present used in the implementation. It is a useful hook left in place to allow a Z with a notation for refinement to be translated by the translator.

A directive could be included in the specification to mark the start of the new design level. When such a directive is encountered, a subordinate table is created. Entities from the specification which are refined in the design step are entered into the new table. When definitions are retrieved from the table, the most recent definition will be returned.

Entities which are not refined in the current design step may be accessed from previous tables.

An abstraction relation which relates the new design level to the previous design level could be included to ensure type correctness of the refinement step.

Changes to the output language

A more radical change to the translator would replace VDM-SL by another output language. Choosing Z as the output language would turn the translator into a pure refinement tool for Z by calculating pre-conditions and expanding schema expressions. This activity should be performed upon the user's direction only. The expansion of all of the schema expressions and the calculation of pre-conditions of all of the operations would greatly increase the length of the specification and mar its clarity.

An alternative output language would be Carroll Morgan's Refinement Calculus. This would allow the refinement to be directed by the laws of the Refinement Calculus.

Simplification of generated predicates

The Z-to-VDM-SL translator performs some simplification of the Z text which is supplied as input to the translator. No attempt is made to simplify the VDM-SL output. In some cases, the simplification of the predicates may be best achieved with human intervention.

This simplification was not considered for the Z-to-VDM-SL translator because a VDM-SL tool is available to perform this simplification (Kneuper, 1987). The reader is reminded that one of the initial guidelines of the project was that it should not contain any significant duplication or re-implementation of any existing Z or VDM-SL tool. Some integration of the Z-to-VDM-SL translator with Kneuper's VDM-SL tool might be required. The implementation work required to integrate the two tools may not be substantial if the VDM-SL tool accepts input in the concrete syntax of (Wolczko, 1990) which is generated by the translator.

Preservation of the intermediate representation

The Z-to-VDM-SL translator creates graphs–representation-independent specifications— as an intermediate representation which is to be transformed before being unparsed as VDM-SL. These intermediate representations are discarded as mere marginal notes in a longer calculation. If preserved, they could be included in an integrated database of specifications stored in a representation-independent form. With appropriate machine support, these specifications could be accessed in one concrete syntax and edited in another without the user needing to know the notation in which they were originally expressed.

14.4.2 Use of machine assistance

For the purposes of this experiment, no machine assistance was employed in the creation of the correctness proofs. This extreme approach was adopted in order to retain fairness in the experiment: no component of the development would be advantaged or disadvantaged by the use of a computer program in its creation. Thus the comparison between developments which employ different levels of rigour remains fair.

A correctness-oriented development which used tools such as interactive proof editors, theorem provers or integrated formal reasoning support environments would provide an interesting alternative approach to this work.

14.5 Strengths of the development

The knowledge that evidence of the fitness of the specification and of the correctness of the implementation may be requested encourages both the specifier and the implementor to simplify and amplify the structure of their work at all times. Thus, the 'correctness-oriented' bias of the development has a beneficial effect on the quality of both the specification and the implementation.

The desire to keep the implementation as simple as possible encourages the implementor to work within the most easily understood subset of the programming language used. This is often the subset of the language which is most reliably implemented by compilers and interpreters and may also be a subset offered by all of the implementations of the language. Thus, the 'correctness-oriented' bias of the development also has a beneficial effect on the quality of the implementation even in cases where the connective tissue of a proof text is not realised.

As CAR Hoare has said, the price to be paid for reliability in software systems is simplicity. The simple aims of the translator project have allowed the project to be completed successfully. It is possible that a more ambitious project in which it was hoped to develop a generic or incremental version of the translator would not have succeeded. Thus, the concern for correctness has had a beneficial influence on the decisions made in the formulation of the initial requirements for the project.

The decision not to include type-checking of Z texts in the translation process necessitates the use of a simple translation strategy. This decision has the added benefit that a partial Z specification can be submitted to the type-checker. A Z specifier can translate a work-in-progress for a VDM-literate colleague without being obliged to turn this into a well-typed Z text.

The lack of some semantic information may mean that the translator will fail to produce an output. However, if an output is produced, it will be an appropriate translation of the input. This indicates that an incremental version of the translator may be produced with relative ease.

The following schema is a sample incomplete Z specification which does not give definitions for the names *Polynomial* or *differentiate*.

Integrate

$p?, p!: Polynomial$

$differentiate(p!) = p?$

The Z-to-VDM-SL translator will assume that *Polynomial* is the name of a type and that *differentiate* is the name of a function. The following VDM-SL operation is produced by the translator.

$Integrate\ (p\text{-}\mathsf{in}: Polynomial)\ p\text{-}\mathsf{out}: Polynomial$

$\mathsf{post}\ differentiate(p\text{-}\mathsf{out}) = p\text{-}\mathsf{in}$

14.6 Shortcomings of the development

14.6.1 Modularity in the development

The interfaces between the modules of the specifications are not well defined. A higher-level module structure such as that used in (Carrington *et al.*, 1989) or (Middelburg, 1989) would have clarified the structure of the specifications.

Such a module structure would not have entirely solved the problem of interfacing modules of the specification which were written in different languages. For the purposes of this development, the *ad hoc* solution of summarizing the imported parts of the modules in the language of the importing module was used.

14.6.2 Treatment of concurrency

No attempt has been made here to analyse the concurrent execution of the implementation. As with the modular composition of the components of the translator, the parallel composition of the components was produced without supporting proof text.

Specifications written in process-oriented specification styles such as CCS (Milner, 1989) or CSP (Hoare, 1985) could be used.

14.7 Context of this work

CAR Hoare's 'worthwhile research project' mentioned at the outset of this thesis remains as interesting a challenge as it was at the time of its proposal. This thesis has shown that a similar project with more modest aims can deliver interesting results about the correctness-oriented development of software systems.

The author encourages readers of this thesis to repeat his experiment with software developments of comparable size or larger. The proficiency with correctness-oriented software development techniques that the author acquires is rare and valuable. A comparison of the results would be of considerable interest.

The degree of rigour employed in the construction of the proofs of correctness of the implementation did not appear to have a significant effect on the correctness of the implementation. Sections defended by sketch proofs did not contain significantly more errors than sections defended by rigorous proofs.

The conclusion to be drawn from this is not that sketch proofs should be provided instead of formal proofs but rather that a software developer can no longer defend not providing small sketch proofs because the budget of the project does not allow all proof obligations to be discharged by fully formal proofs.

14.8 Software engineering as science

There is scant enough beauty in the contemporary practice of software engineering that we can afford to ignore it when it is found. It is for this reason that we must continue to study correctness. Our simple models of computability, specification and refinement are all that we have to guide us through a confusing maze of notations, paradigms and methodologies.

Proofs are useful programming tools. They should be as familiar a part of a programmer's toolset as an editor, a compiler or a symbolic debugger. The terminology of set-theoretic mathematics should be as much a part of the language of a software developer as the terminology of formal languages, operating systems and program assembly.

Software engineering is a mathematical science with laws of unusual beauty and simplicity. Many software developers are at present ignorant even of the existence of these laws let alone their character or their applicability. It is profoundly to be wished that this situation should change.

But for us now
The beyond is still out there as on tiptoes here we stand
On promontories that are themselves a-tiptoe
Reluctant to be land.

from Western Landscape
by Louis MacNeice

Appendix A

The string classification function

This function is a finite injective partial function from character sequences to terminal symbol classes.

$$
\begin{array}{l}
Terminal\_Symbol\_Table == \\
\quad \{\text{`}V\text{'} \mapsto Start\_Version, \\
\quad \text{`}Z\text{'} \mapsto Start\_Z, \\
\quad \text{`}EI\text{'} \mapsto End\_Indent, \\
\quad \text{`}ER\text{'} \mapsto End\_Rule, \\
\quad \text{`}EV\text{'} \mapsto End\_Version, \\
\quad \text{`}EZ\text{'} \mapsto End\_Z, \\
\quad \text{`}GE\text{'} \mapsto Generic\_Def1, \\
\quad \text{`}NL\text{'} \mapsto New\_Line, \\
\quad \text{`}Pi\text{'} \mapsto Pi, \\
\quad \text{`}SB\text{'} \mapsto Start\_Schema, \\
\quad \text{`}SI\text{'} \mapsto Start\_Indent, \\
\quad \text{`}SR\text{'} \mapsto Set\_Rule, \\
\quad \text{`}ST\text{'} \mapsto Middle\_Schema, \\
\quad \text{`}TH\text{'} \mapsto Start\_Theorem, \\
\quad \text{`}Xi\text{'} \mapsto Xi, \\
\quad \text{`}id\text{'} \mapsto Ident, \\
\quad \text{`}in\text{'} \mapsto In\_Seq, \\
\quad \text{`}mu\text{'} \mapsto Mu, \\
\quad \text{`}or\text{'} \mapsto Or\_Sy, \\
\quad \text{`}ESB\text{'} \mapsto End\_Schema, \\
\quad \text{`}ETH\text{'} \mapsto End\_Theorem, \\
\quad \text{`}Nat\text{'} \mapsto Nat, \\
\quad \text{`}all\text{'} \mapsto All\_Sy, \\
\quad \text{`}and\text{'} \mapsto And\_Sy, \\
\quad \text{`}bij\text{'} \mapsto Bij, \\
\quad \text{`}cat\text{'} \mapsto Seq\_Conc, \\
\quad \text{`}cmp\text{'} \mapsto Rel\_Comp, \\
\quad \text{`}def\text{'} \mapsto Term\_Syn\_Eq, \\
\quad \text{`}div\text{'} \mapsto Div\_Sy, \\
\quad \text{`}dlr\text{'} \mapsto Dollar, \\
\quad \text{`}dom\text{'} \mapsto Dom,
\end{array}
$$

$`dot' \mapsto Such\_That,$
$`exi' \mapsto Exists,$
$`geq' \mapsto Geq,$
$`iff' \mapsto Equiv,$
$`imp' \mapsto Imp,$
$`int' \mapsto Inter,$
$`inv' \mapsto Inv,$
$`leq' \mapsto Leq,$
$`map' \mapsto Maps\_To,$
$`max' \mapsto Max,$
$`mem' \mapsto In\_Sy,$
$`min' \mapsto Min,$
$`mod' \mapsto Mod\_Sy,$
$`nem' \mapsto Not\_In,$
$`neq' \mapsto Not\_Eq,$
$`not' \mapsto Not\_Sy,$
$`pre' \mapsto Z\_Pre,$
$`ran' \mapsto Ran,$
$`rel' \mapsto Rel,$
$`rev' \mapsto Reverse\_Sy,$
$`seq' \mapsto Seq,$
$`sub' \mapsto Start\_Sub,$
$`sup' \mapsto Start\_Super,$
$`tcl' \mapsto T\_Closure,$
$`uni' \mapsto Union,$
$`zeq' \mapsto Z\_Eq,$
$`zor' \mapsto Z\_Or,$
$`bbar' \mapsto Branch\_Sep,$
$`cbar' \mapsto Constraint,$
$`dcat' \mapsto Dist\_Conc,$
$`dcmp' \mapsto Dist\_Comp,$
$`ddef' \mapsto Data\_Type\_Def,$
$`diff' \mapsto Set\_Diff,$
$`dint' \mapsto Dist\_Inter,$
$`dovr' \mapsto Dist\_Over,$
$`dres' \mapsto Dom\_Rest,$
$`dsub' \mapsto Dom\_Sub,$
$`duni' \mapsto Dist\_Union,$
$`esub' \mapsto End\_Subs,$
$`esup' \mapsto End\_Super,$
$`exi1' \mapsto Exists1,$
$`fcmp' \mapsto Forward\_Rel\_Comp,$
$`ffun' \mapsto Finite\_Func,$
$`finj' \mapsto Finite\_Inj,$
$`fovr' \mapsto Func\_Over,$
$`fset' \mapsto Finite\_Subs,$
$`head' \mapsto Head,$

$$
\begin{aligned}
&\text{‘}ires\text{’} \mapsto Index\_Rest,\\
&\text{‘}iter\text{’} \mapsto Iter,\\
&\text{‘}lang\text{’} \mapsto Left\_Seq,\\
&\text{‘}last\text{’} \mapsto Last,\\
&\text{‘}limg\text{’} \mapsto Left\_Image,\\
&\text{‘}lsch\text{’} \mapsto Left\_Schema,\\
&\text{‘}lseq\text{’} \mapsto Start\_Seq,\\
&\text{‘}lset\text{’} \mapsto Left\_Set,\\
&\text{‘}lsqb\text{’} \mapsto Left\_Square,\\
&\text{‘}next\text{’} \mapsto Next,\\
&\text{‘}null\text{’} \mapsto Null\_Set,\\
&\text{‘}pfun\text{’} \mapsto Partial\_Func,\\
&\text{‘}pinj\text{’} \mapsto Partial\_Inj,\\
&\text{‘}pred\text{’} \mapsto Pred\_Sy,\\
&\text{‘}prod\text{’} \mapsto Cart\_Prod,\\
&\text{‘}pset\text{’} \mapsto Powerset,\\
&\text{‘}psur\text{’} \mapsto Partial\_Sur,\\
&\text{‘}rang\text{’} \mapsto Right\_Seq,\\
&\text{‘}rimg\text{’} \mapsto Right\_Image,\\
&\text{‘}rres\text{’} \mapsto Range\_Rest,\\
&\text{‘}rsch\text{’} \mapsto Right\_Schema,\\
&\text{‘}rseq\text{’} \mapsto End\_Seq,\\
&\text{‘}rset\text{’} \mapsto Right\_Set,\\
&\text{‘}rsqb\text{’} \mapsto Right\_Square,\\
&\text{‘}rsub\text{’} \mapsto Range\_Sub,\\
&\text{‘}rtcl\text{’} \mapsto R\_T\_Closure,\\
&\text{‘}sdef\text{’} \mapsto Schema\_Syn\_Eq,\\
&\text{‘}seq1\text{’} \mapsto Seq1,\\
&\text{‘}sres\text{’} \mapsto Seq\_Restrict,\\
&\text{‘}subs\text{’} \mapsto Subset,\\
&\text{‘}succ\text{’} \mapsto Succ\_Sy,\\
&\text{‘}tail\text{’} \mapsto Tail,\\
&\text{‘}tfun\text{’} \mapsto Total\_Func,\\
&\text{‘}thrm\text{’} \mapsto Theorem,\\
&\text{‘}tinj\text{’} \mapsto Total\_Inj,\\
&\text{‘}tsur\text{’} \mapsto Total\_Sur,\\
&\text{‘}zall\text{’} \mapsto Z\_For\_All,\\
&\text{‘}zand\text{’} \mapsto Z\_And,\\
&\text{‘}zcmp\text{’} \mapsto Z\_Comp,\\
&\text{‘}zexi\text{’} \mapsto Z\_Exists,\\
&\text{‘}zfor\text{’} \mapsto Z\_Rename,\\
&\text{‘}zimp\text{’} \mapsto Z\_Imp,\\
&\text{‘}znot\text{’} \mapsto Z\_Not,\\
&\text{‘}zovr\text{’} \mapsto Z\_Over,\\
&\text{‘}Delta\text{’} \mapsto Delta\_Sy,\\
&\text{‘}Sigma\text{’} \mapsto Sigma,\\
&\text{‘}front\text{’} \mapsto Front,
\end{aligned}
$$

$$
\begin{aligned}
&\text{`}\mathit{fset1}\text{'} \mapsto \mathit{Finite\_Subset1}, \\
&\text{`}\mathit{pset1}\text{'} \mapsto \mathit{Powerset1}, \\
&\text{`}\mathit{psubs}\text{'} \mapsto \mathit{Proper\_Subset}, \\
&\text{`}\mathit{riter}\text{'} \mapsto \mathit{End\_Iter}, \\
&\text{`}\mathit{theta}\text{'} \mapsto \mathit{Theta}, \\
&\text{`}\mathit{where}\text{'} \mapsto \mathit{Where}, \\
&\text{`}\mathit{zhide}\text{'} \mapsto \mathit{Z\_Hide}, \\
&\text{`}\mathit{zpipe}\text{'} \mapsto \mathit{Z\_Pipe}, \\
&\text{`}\mathit{lambda}\text{'} \mapsto \mathit{Lambda}, \\
&\text{`}\mathit{prefix}\text{'} \mapsto \mathit{Prefix}, \\
&\text{`}\mathit{squash}\text{'} \mapsto \mathit{Squash}, \\
&\text{`}\mathit{suffix}\text{'} \mapsto \mathit{Suffix}, \\
&\text{`}\mathit{monotonic}\text{'} \mapsto \mathit{Mono}, \\
&\text{`}\mathit{total\_order}\text{'} \mapsto \mathit{Total\_Ord}, \\
&\text{`}\mathit{partial\_order}\text{'} \mapsto \mathit{Part\_Ord}\}
\end{aligned}
$$

Appendix B

Syntax of Z

The syntax of the Z input language of the specification source-to-source translator is now given. Production rules are given in alphabetical order of the name of the syntactic construct which appears on the left hand side of the rule. Explanatory comments appear on the right.

Optional constructs are enclosed in angle brackets ($\langle\rangle$). Visual formatting commands such as instructions for new lines and indentation and names for symbols not in this character set are shown in bold. The distinguished (origin) symbol is *z_script*.

axiomatic_def	$::=$ **set_rule** $\langle$*parameters* **generic_def**1$\rangle$ *schema_text* **end_rule**	
dec	$::=$ *id_list* $\langle$: *term*$\rangle$	variable declarations
dec_list	$::=$ *dec* $\langle$*list_sep dec_list*$\rangle$	
definition	$::=$ *axiomatic_def*	
	$\mid$ *schema_def*	
digit	$::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$	
exp	$::=$ *product_list* $\langle$*infix_generic exp*$\rangle$	expressions
exp_0	$::= \lambda$ *schema_text* $\cdot$ *exp*	
	$\mid\ \mu$ *schema_text* $\cdot$ *exp*	
	$\mid$ *exp*	
exp_1	$::=$ *exp_2* $\langle$*infix_func_1 exp_1*$\rangle$	
exp_2	$::=$ *exp_3* $\langle$*infix_func_2 exp_2*$\rangle$	
exp_3	$::=$ *exp_4* $\langle$*infix_func_3 exp_3*$\rangle$	
exp_4	$::=$ *exp_5* $\langle$*infix_func_4 exp_4*$\rangle$	
exp_5	$::=$ *exp_6* $\langle$*infix_func_5 exp_5*$\rangle$	

exp_6	$::= exp\_7 \ \langle infix\_func\_6 \ exp\_6 \rangle$	
exp_7	$::= \langle pre\_gen\_seq \rangle \ exp\_8$	
exp_8	$::= exp\_9 \ \langle exp\_8\_param \rangle$	function application
exp_8_param	$::= (exp\_list)$	function parameter list
	$\mid \ (\!\mid \ exp\_0 \ \mid\!)$	relational image
exp_9	$::= exp\_10 \ \langle post\_fun\_seq \rangle$	
exp_10	$::= id \ \langle instantiation \rangle$	
	$\mid \ natural\_number$	
	$\mid \ set\_exp$	
	$\mid \ left\_seq \ \langle exp\_list \rangle \ right\_seq$	
	$\mid \ (exp\_list)$	tuples or expressions in parentheses
exp_list	$::= exp\_0 \ \langle , exp\_list \rangle$	an expression list
$given\_set\_def$	$::= [id\_list]$	assumed sets
id	$::=$ "alphanumeric characters"	an identifier
id_list	$::= id \ \langle , id\_list \rangle$	a list of identifiers
$infix\_func\_1$	$::= \mapsto$	the 'maps to' symbol
$infix\_func\_2$	$::= \ldots$	the 'up to' symbol
$infix\_func\_3$	$::= +$	plus
	$\mid \ -$	minus
	$\mid \ \cup$	set union
	$\mid \ \setminus$	set difference
	$\mid \ \frown$	sequence catenation
$infix\_func\_4$	$::= *$	times
	$\mid$ div	div as in Pascal
	$\mid$ mod	mod as in Pascal
	$\mid \ \cap$	set intersection
	$\mid \ \fatsemi$	forward composition
	$\mid \ \circ$	backward composition
	$\mid \ \upharpoonright$	sequence filtering
$infix\_func\_5$	$::= \oplus$	function overriding
$infix\_func\_6$	$::= \lhd$	domain restriction

	$\vert$	$\rhd$	range restriction
	$\vert$	$\ndres$	domain subtraction
	$\vert$	$\nrres$	range subtraction
infix_generic	$::=$	$\pfun$	partial function
	$\vert$	$\fun$	total function
	$\vert$	$\ffun$	finite function
	$\vert$	$\pinj$	partial injection
	$\vert$	$\inj$	total injection
	$\vert$	$\finj$	finite injection
	$\vert$	$\psurj$	partial surjection
	$\vert$	$\surj$	total surjection
	$\vert$	$\bij$	bijection
instantiation	$::=$	[*exp_list*]	
left_seq	$::=$	$\langle$	
list_sep	$::=$	;	
	$\vert$	**new_line**	
natural_number	$::=$	*digit* $\langle$*natural_number*$\rangle$	
parameters	$::=$	[*id_list*]	
post_fun	$::=$	$^\sim$	inverse ($f^\sim \equiv f^{-1}$)
	$\vert$	*	reflexive transitive closure
	$\vert$	$^+$	transitive closure
	$\vert$	**iter** *exp* **end_iter**	function iteration
post_fun_seq	$::=$	*post_fun* $\langle$*post_fun_seq*$\rangle$	
pre_gen	$::=$	$\mathbb{P}$	powerset
	$\vert$	$\mathbb{P}_1$	non-empty powerset
	$\vert$	$\mathbb{F}$	finite subsets
	$\vert$	$\mathbb{F}_1$	non-empty finite subsets
	$\vert$	id	identity relation
	$\vert$	seq	finite sequences
	$\vert$	seq_1	finite non-empty sequences
	$\vert$	$\bigcup$	distributed intersection
	$\vert$	$\bigcap$	distributed union
	$\vert$	dom	domain of a relation
	$\vert$	ran	range of a relation
	$\vert$	#	cardinality of a set
	$\vert$	$\frown /$	distributed concatenation
	$\vert$	$-$	unary minus
	$\vert$	head	head of a sequence

	$\mid$	tail	decapitated sequence
	$\mid$	last	foot of a sequence
	$\mid$	front	depedicated sequence
	$\mid$	reverse	sequence reverse
pre_gen_seq	$::=$	*pre_gen* $\langle$*pre_gen_seq*$\rangle$	
pred	$::=$	$\exists$ *schema_text* $\cdot$ *pred*	
	$\mid$	$\exists_1$ *schema_text* $\cdot$ *pred*	
	$\mid$	$\forall$ *schema_text* $\cdot$ *pred*	
	$\mid$	*pred_1*	
pred_1	$::=$	*pred_2* $\langle\Leftrightarrow$ *pred_1*$\rangle$	
pred_2	$::=$	*pred_3* $\langle\Rightarrow$ *pred_2*$\rangle$	
pred_3	$::=$	*pred_4* $\langle\vee$ *pred_3*$\rangle$	
pred_4	$::=$	*pred_5* $\langle\wedge$ *pred_4*$\rangle$	
pred_5	$::=$	$\neg$ *exp* $\langle$*rel exp*$\rangle$	
pred_list	$::=$	*pred* $\langle$*list_sep pred_list*$\rangle$	
product_list	$::=$	*exp_1* $\langle\times$*product_list*$\rangle$	
rel	$::=$	$=$	equality
	$\mid$	$\in$	set membership
	$\mid$	$\neq$	inequality
	$\mid$	$\notin$	set absence
	$\mid$	$\subset$	proper subset
	$\mid$	$\subseteq$	subset
	$\mid$	$<$	less than
	$\mid$	$>$	greater than
	$\mid$	$\geqslant$	greater than or equal to
	$\mid$	$\leqslant$	less than or equal to
right_seq	$::=$	$\rangle$	
schema_body	$::=$	$\widehat{=}$ *schema_exp*	
	$\mid$	**start_schema** *schema_text* **end_schema**	
schema_def	$::=$	*id* $\langle$*parameters*$\rangle$ *schema_body*	
schema_exp	$::=$	$\forall$ *schema_text* $\cdot$ *schema_exp*	

	$\mid$	$\exists$ *schema_text* $\cdot$ *schema_exp*	
	$\mid$	*schema_exp_1*	
schema_exp_1	$::=$	*schema_exp_2* $\langle\Leftrightarrow$ *schema_exp_1*$\rangle$	
schema_exp_2	$::=$	*schema_exp_3* $\langle\Rightarrow$ *schema_exp_2*$\rangle$	
schema_exp_3	$::=$	*schema_exp_4* $\langle\vee$ *schema_exp_3*$\rangle$	
schema_exp_4	$::=$	*schema_exp_5* $\langle\wedge$ *schema_exp_4*$\rangle$	
schema_exp_5	$::=$	*schema_exp_6* $\langle\fatsemi$ *schema_exp_5*$\rangle$	sequential composition
schema_exp_6	$::=$	*schema_exp_7* $\langle\setminus$ *schema_exp_6*$\rangle$	variable hiding
schema_exp_7	$::=$	$\neg$ *schema_exp_7*	schema negation
	$\mid$	pre *schema_exp_7*	schema pre-condition
	$\mid$	*schema_ref*	schema reference
schema_ref	$::=$	*id* $\langle$*instantiation*$\rangle$	
schema_text	$::=$	*dec_list* $\langle$**middle_schema** *pred_list*$\rangle$	constrained declarations
set_body	$::=$	*exp_list*	set described in extension
	$\mid$	. *schema_text* $\langle\cdot$ *exp*$\rangle$.	set described by predicate and expression
set_exp	$::=$	$\varnothing$ $\langle[$*exp*$]\rangle$	the empty set
	$\mid$	$\mathbb{N}$	the natural numbers
	$\mid$	$\{$*set_body*$\}$	a set literal
z_phrase	$::=$	*given_set_def*	
	$\mid$	*definition*	
z_script	$::=$	**start_z** *z_text* **end_z** $\langle$*z_script*$\rangle$	
	$\mid$	“explanatory text” $\langle$*z_script*$\rangle$	
z_text	$::=$	*z_phrase* $\langle$*list_sep z_text*$\rangle$	

Appendix C

Output of the Z-to-VDM-SL Translator

This appendix contains a VDM-SL translation of the initial Z specification of a source handler presented in Chapter 6. The translation was performed by the Z-to-VDM-SL translator. The specification appears here exactly as it was produced by the translator.

* * *

A simple source handler is required to read a Z document, making available characters labelled with their row and column position. The source handler should be a reusable software unit, for use in lexical analyzers, syntax analyzers and other specification processing tools.

C.1 Specification

The source handler must recognize end of line and end of file characters. Upon detecting the end of line character, the current row position is incremented and the current column position reset. The end of file character is generated by the source handler once the Z document is exhausted. The end of line and end of file characters are distinct. A set of characters, $CHAR$, is assumed.

$\mathsf{assume}\ CHAR;$

$end\text{-}of\text{-}line\colon\ \to CHAR$
$\mathsf{post}\ end\text{-}of\text{-}line \neq end\text{-}of\text{-}file$

$end\text{-}of\text{-}file\colon\ \to CHAR$
$\mathsf{post}\ end\text{-}of\text{-}line \neq end\text{-}of\text{-}file$

The source handler accesses the Z document and maintains the current row and column position in the document. The Z document is a—possibly empty—sequence of characters. The row and column position are positive integers.

$$\begin{array}{ll} State\text{-}\mathsf{type} :: & document\colon CHAR^*; \\ & row, column\colon \mathbb{N} \end{array}$$

where

$$inv\text{-}State\text{-}\mathsf{type}(mk\text{-}State\text{-}\mathsf{type}(document, row, column)) \quad \triangleq$$
$$row \neq 0 \land$$
$$column \neq 0$$

The initial state of the source handler requires the row and column positions to be 1 and requires that the Z document does not contain the *end-of-file* character. This restriction is necessary since the *end-of-file* character is used as a special character in the specification and is understood to be recognized as a distinguished member of the *CHAR* set.

$State_0$ ()
ext wr $document \colon CHAR^*$
  wr $row, column \colon \mathbb{N}$
post $row = 1 \land$
  $column = 1 \land$
  $end\text{-}of\text{-}file \notin \mathsf{rng}\ document$

The source handler produces document characters which contain a character from the document and its document position. These are now formally described.

$$Document\text{-}Character\text{-}\mathsf{type} :: ch \colon CHAR; \quad row, column \colon \mathbb{N}$$

where

$$inv\text{-}Document\text{-}Character\text{-}\mathsf{type}(mk\text{-}Document\text{-}Character\text{-}\mathsf{type}(ch, row, column)) \quad \triangleq$$
$$row \neq 0 \land$$
$$column \neq 0$$

C.1.1 Operation of the source handler

The source handler provides a single operation to make the next document character available. The three possible cases to be considered when supplying the next character relate to:

1. reaching the end of the document;
2. reaching the end of a line in the document; and
3. returning a single character.

In every case a character is provided whose document position is determined by the row and column values in the present state.

Establish-Position () $doc\text{-}char\text{-}\mathsf{out} \colon Document\text{-}Character$
ext wr $State \colon State\text{-}\mathsf{type}$
post let $State = mk\text{-}State\text{-}\mathsf{type}(document, row, column)$ in
  let $\overleftarrow{State} = mk\text{-}State\text{-}\mathsf{type}(\overleftarrow{document}, \overleftarrow{row}, \overleftarrow{column})$ in
  $row(doc\text{-}char\text{-}\mathsf{out}) = \overleftarrow{row} \land$
  $column(doc\text{-}char\text{-}\mathsf{out}) = \overleftarrow{column}$

For the cases where the next document character is provided, it is useful to define an operation which makes this character available.

$$
\begin{array}{l}
\textit{Obtain-Character}\ ()\ \textit{doc-char}\text{-}\mathsf{out}\colon \textit{Document-Character}\\
\mathsf{ext\ wr}\ \textit{State}\colon \textit{State}\text{-}\mathsf{type}\\
\mathsf{post\ let}\ \textit{State} = \textit{mk-State}\text{-}\mathsf{type}(\textit{document}, \textit{row}, \textit{column})\ \mathsf{in}\\
\qquad \mathsf{let}\ \overleftarrow{\textit{State}} = \textit{mk-State}\text{-}\mathsf{type}(\overleftarrow{\textit{document}}, \overleftarrow{\textit{row}}, \overleftarrow{\textit{column}})\ \mathsf{in}\\
\qquad \overleftarrow{\textit{document}} = [\textit{ch}(\textit{doc-char}\text{-}\mathsf{out})] \curvearrowright \textit{document}
\end{array}
$$

The actions taken for each case are now defined.

The *At-End-Of-File* operation

The following schema defines the action to be taken upon reaching the end of the document sequence. The end of file character is passed on. The state of the source handler does not change.

$$
\begin{array}{l}
\textit{At-End-Of-File}\ ()\ \textit{doc-char}\text{-}\mathsf{out}\colon \textit{Document-Character}\\
\mathsf{ext\ wr}\ \textit{State}\colon \textit{State}\text{-}\mathsf{type}\\
\mathsf{pre\ let}\ \textit{State} = \textit{mk-State}\text{-}\mathsf{type}(\textit{document}, \textit{row}, \textit{column})\ \mathsf{in}\\
\quad \textit{document} = []\\
\mathsf{post\ let}\ \textit{State} = \textit{mk-State}\text{-}\mathsf{type}(\textit{document}, \textit{row}, \textit{column})\ \mathsf{in}\\
\qquad \mathsf{let}\ \overleftarrow{\textit{State}} = \textit{mk-State}\text{-}\mathsf{type}(\overleftarrow{\textit{document}}, \overleftarrow{\textit{row}}, \overleftarrow{\textit{column}})\ \mathsf{in}\\
\qquad \textit{State} = \overleftarrow{\textit{State}} \wedge\\
\qquad \textit{ch}(\textit{doc-char}\text{-}\mathsf{out}) = \textit{end-of-file} \wedge\\
\qquad \textit{row}(\textit{doc-char}\text{-}\mathsf{out}) = \overleftarrow{\textit{row}} \wedge\\
\qquad \textit{column}(\textit{doc-char}\text{-}\mathsf{out}) = \overleftarrow{\textit{column}}
\end{array}
$$

The *At-End-Of-Line* operation

At the end of a line of the Z input document, the source handler returns the end of line character with the corresponding document position. The end of line character is removed from the document, the current row number is incremented and the current column position reset to 1.

$$
\begin{array}{l}
\textit{At-End-Of-Line}\ ()\ \textit{doc-char}\text{-}\mathsf{out}\colon \textit{Document-Character}\\
\mathsf{ext\ wr}\ \textit{State}\colon \textit{State}\text{-}\mathsf{type}\\
\mathsf{pre\ let}\ \textit{State} = \textit{mk-State}\text{-}\mathsf{type}(\textit{document}, \textit{row}, \textit{column})\ \mathsf{in}\\
\quad \mathsf{hd}\ \textit{document} = \textit{end-of-line}\\
\mathsf{post\ let}\ \textit{State} = \textit{mk-State}\text{-}\mathsf{type}(\textit{document}, \textit{row}, \textit{column})\ \mathsf{in}\\
\qquad \mathsf{let}\ \overleftarrow{\textit{State}} = \textit{mk-State}\text{-}\mathsf{type}(\overleftarrow{\textit{document}}, \overleftarrow{\textit{row}}, \overleftarrow{\textit{column}})\ \mathsf{in}\\
\qquad \textit{row} = \overleftarrow{\textit{row}} + 1 \wedge\\
\qquad \textit{column} = 1 \wedge\\
\qquad \textit{row}(\textit{doc-char}\text{-}\mathsf{out}) = \overleftarrow{\textit{row}} \wedge\\
\qquad \textit{column}(\textit{doc-char}\text{-}\mathsf{out}) = \overleftarrow{\textit{column}} \wedge\\
\qquad \overleftarrow{\textit{document}} = [\textit{ch}(\textit{doc-char}\text{-}\mathsf{out})] \curvearrowright \textit{document}
\end{array}
$$

The *At-Next-Character* operation

If no end of line character has been detected and the document is not exhausted, the next character of the document is returned. It is removed from the beginning of the document and the column position is incremented.

$$
\begin{array}{l}
\mathit{At\text{-}Next\text{-}Character}\ ()\ \mathit{doc\text{-}char}\text{-}\mathsf{out}\colon \mathit{Document\text{-}Character}\\
\mathsf{ext\ wr}\ \mathit{State}\colon \mathit{State}\text{-}\mathsf{type}\\
\mathsf{pre\ let}\ \mathit{State} = \mathit{mk\text{-}State}\text{-}\mathsf{type}(\mathit{document}, \mathit{row}, \mathit{column})\ \mathsf{in}\\
\quad \mathsf{hd}\ \mathit{document} \neq \mathit{end\text{-}of\text{-}line}\\
\mathsf{post\ let}\ \mathit{State} = \mathit{mk\text{-}State}\text{-}\mathsf{type}(\mathit{document}, \mathit{row}, \mathit{column})\ \mathsf{in}\\
\qquad \mathsf{let}\ \overleftarrow{\mathit{State}} = \mathit{mk\text{-}State}\text{-}\mathsf{type}(\overleftarrow{\mathit{document}}, \overleftarrow{\mathit{row}}, \overleftarrow{\mathit{column}})\ \mathsf{in}\\
\qquad \mathit{row} = \overleftarrow{\mathit{row}} \wedge\\
\qquad \mathit{column} = \overleftarrow{\mathit{column}} + 1 \wedge\\
\qquad \mathit{row}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \overleftarrow{\mathit{row}} \wedge\\
\qquad \mathit{column}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \overleftarrow{\mathit{column}} \wedge\\
\qquad \overleftarrow{\mathit{document}} = [\mathit{ch}(\mathit{doc\text{-}char}\text{-}\mathsf{out})] \curvearrowright \mathit{document}
\end{array}
$$

The behaviour of the source handler is different for each of the three cases.

$$
\begin{array}{l}
\mathit{Read\text{-}Character}\ ()\ \mathit{doc\text{-}char}\text{-}\mathsf{out}\colon \mathit{Document\text{-}Character}\\
\mathsf{ext\ wr}\ \mathit{State}\colon \mathit{State}\text{-}\mathsf{type}\\
\mathsf{post\ let}\ \mathit{State} = \mathit{mk\text{-}State}\text{-}\mathsf{type}(\mathit{document}, \mathit{row}, \mathit{column})\ \mathsf{in}\\
\qquad \mathsf{let}\ \overleftarrow{\mathit{State}} = \mathit{mk\text{-}State}\text{-}\mathsf{type}(\overleftarrow{\mathit{document}}, \overleftarrow{\mathit{row}}, \overleftarrow{\mathit{column}})\ \mathsf{in}\\
\qquad (\mathit{State} = \overleftarrow{\mathit{State}} \wedge\\
\qquad \overleftarrow{\mathit{document}} = [\,] \wedge\\
\qquad \mathit{ch}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \mathit{end\text{-}of\text{-}file} \wedge\\
\qquad \mathit{row}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \overleftarrow{\mathit{row}} \wedge\\
\qquad \mathit{column}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \overleftarrow{\mathit{column}}) \vee\\
\qquad ((\mathsf{hd}\ \overleftarrow{\mathit{document}} = \mathit{end\text{-}of\text{-}line} \wedge\\
\qquad \mathit{row} = \overleftarrow{\mathit{row}} + 1 \wedge\\
\qquad \mathit{column} = 1 \wedge\\
\qquad \mathit{row}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \overleftarrow{\mathit{row}} \wedge\\
\qquad \mathit{column}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \overleftarrow{\mathit{column}} \wedge\\
\qquad \overleftarrow{\mathit{document}} = [\mathit{ch}(\mathit{doc\text{-}char}\text{-}\mathsf{out})] \curvearrowright \mathit{document}) \vee\\
\qquad (\mathsf{hd}\ \overleftarrow{\mathit{document}} \neq \mathit{end\text{-}of\text{-}line} \wedge\\
\qquad \mathit{row} = \overleftarrow{\mathit{row}} \wedge\\
\qquad \mathit{column} = \overleftarrow{\mathit{column}} + 1 \wedge\\
\qquad \mathit{row}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \overleftarrow{\mathit{row}} \wedge\\
\qquad \mathit{column}(\mathit{doc\text{-}char}\text{-}\mathsf{out}) = \overleftarrow{\mathit{column}} \wedge\\
\qquad \overleftarrow{\mathit{document}} = [\mathit{ch}(\mathit{doc\text{-}char}\text{-}\mathsf{out})] \curvearrowright \mathit{document}))
\end{array}
$$

Bibliography

Abrial, J.-R. 1988. The B Tool: Extended Abstract. *In: VDM 88: VDM—The Way Ahead.* Springer-Verlag.

Achour, Said. 1988a. *A Comparison of the Proof Techniques in Two Formal Specification Methodologies.* Tech. rept. CS-88-25. Department of Computer Science, University of Sheffield.

Achour, Said. 1988b. *A Comparison of Two Constructive Approaches to Formal Specification.* Tech. rept. CS-88-16. Department of Computer Science, University of Sheffield.

Barnes, John G.P. 1989. *Programming in Ada.* Third edn. International Computer Science Series. Addison-Wesley.

Barrett, Geoff. 1988. Formal Methods Applied to a Floating Point Number System. *IEEE Transactions on Software Engineering*, May, 611–621.

Bjørner, D., & Jones, C.B. (eds). 1978. *The Vienna Development Method: The Meta-Language.* Lecture Notes in Computer Science, vol. 61. Springer-Verlag.

Bowen, Jonathan (ed). 1987 (Dec.). *Proceedings of Z Users Meeting.*

Broy, Manfred. 1983. Algebraic methods for program construction: The project CIP. *In: Program Transformation and Programming Environments.* NATO.

Burstall, Rod M., & Goguen, Joseph A. 1980. The Semantics of Clear, a Specification Language. *In: Proceedings of Advanced Course on Abstract Software Specifications.* Copenhagen: Springer-Verlag.

Carrington, D., Duke, D., Duke, R., King, P., Rose, G., & Smith, G. 1989 (July). *Formal Specification in Object-Z: Introduction and Case Studies.* Tech. rept. 105. Department of Computer Science, University of Queensland.

Chandy, K. Mani, & Misra, Jayadev. 1988. *Parallel Program Design: A Foundation.* Addison-Wesley.

Clocksin, W.F., & Mellish, C.S. 1987. *Programming in Prolog.* Third edn. Springer-Verlag.

Cohen, B., Harwood, W.T., & Jackson, M.I. 1986. *The Specification of Complex Systems.* Addison-Wesley.

Duce, D.A., & Fielding, E.V.C. 1987. Formal Specification—A Comparison of Two Techniques. *The Computer Journal*, **30**(4), 316–327.

Gehani, Narain. 1984. *Ada—An Advanced Introduction Including Reference Manual for the Ada Programming Language.* Software Series. Prentice-Hall.

Gilmore, S., & Clint, M. 1987 (July). *The Formal Specification of Sorting Algorithms: A Tutorial Exercise.*

Goguen, Joseph A., & Tardo, J. 1979. An introduction to OBJ: a language for writing and testing software specifications. *Pages 170–189 of: Specification of Reliable Software.* IEEE.

Gries, David. 1981. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag.

Hayes, I. (ed). 1987. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall.

Hehner, E.C.R. 1984. *The Logic of Programming.* International Series in Computer Science. Prentice-Hall.

Hoare, C.A.R. 1971. Proof of a Program: FIND. *Communications of the ACM*, **14**(Jan.), 39–45.

Hoare, C.A.R. 1972. Proof of a structured program: The Sieve of Eratosthenes. *BCS Computer Journal*, **15**(Nov.), 321–5.

Hoare, C.A.R. 1985. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall.

Hoare, C.A.R. 1986. Mathematics of Programming. *Byte*, Aug.

Jensen, K., & Wirth, N. 1985. *Pascal: User Manual and Report.* Third edn. Springer-Verlag.

Jones, C.B. 1980. *Software Development: A Rigorous Approach.* International Series in Computer Science. Prentice-Hall.

Jones, C.B. 1990. *Systematic Software Development Using VDM.* Second edn. International Series in Computer Science. Prentice-Hall.

Kilpatrick, P., McParland, P., & Gilmore, S. 1990. The Formal Development of Robot Software. *In: IEE Computing and Control Division Colloquium on The Application of CASE Tools.* London: The Institution of Electrical Engineers.

King, Steve, Sørensen, Ib, & Woodcock, Jim. 1988 (July). *Z: Grammar and Concrete and Abstract Syntaxes.* Tech. rept. PRG-68. Programming Research Group, Oxford.

Kneuper, Ralf. 1987. *Symbolic Execution of Specifications: User Interface and Scenarios.* Tech. rept. UMCS-87-12-6. Department of Computer Science, University of Manchester.

Lano, K., & Breuer, P.T. 1989. From Programs to Z Specifications. *In: Proceedings of the Fourth Annual Z Users Meeting.* Programming Research Group, Oxford.

McGettrick, Andrew D. 1982. *Program Verification Using Ada.* Cambridge Computer Science Texts, vol. 13. Cambridge University Press.

McParland, Patrick J. 1989 (Oct.). *Software Tools to Support Formal Methods.* Ph.D. thesis, Department of Computer Science, The Queen's University of Belfast.

Middelburg, C.A. 1989. VVSL: A language for structured VDM specifications. *Formal Aspects of Computer Science*, **1**(1).

Milner, R. 1989. *Communication and Concurrency.* International Series in Computer Science. Prentice-Hall.

Morgan, C. 1990. *Programming from Specifications.* International Series in Computer Science. Prentice-Hall.

Nipkow, T.N. 1986. Mule: Persistence and Types in an IPSE. *In: Mule—An Environment for Rigorous Software Development.* Department of Computer Science, University of Manchester.

Sannella, D., & Wirsing, M. 1983. A Kernel Language for Algebraic Specification and Implementation. *In: International Conference on Foundations of Computation Theory.* Springer-Verlag.

Sannella, Don. 1988 (July). *A Survey of Formal Software Development Methods.* Tech. rept. ECS-LFCS-88-56. Laboratory for Foundations of Computer Science, University of Edinburgh.

Spivey, J.M. 1988a. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science, vol. 3. Cambridge University Press.

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

Spivey, Mike. 1988b. *The fUZZ Manual.* J.M. Spivey Computing Science Consultancy.

Stavridou, V., Barringer, H., & Edwards, D. 1987. *Formal Specification and Verification of Hardware: A Comparitive Case Study.* Tech. rept. UMCS-87-11-1. Department of Computer Science, University of Manchester.

Sufrin, Bernard. 1982. Formal Specification of a Display-Oriented Text Editor. *Science of Computer Programming*, **1**(2).

Wikström, Åke. 1987. *Functional Programming using Standard ML.* International Series in Computer Science. Prentice-Hall.

Wolczko, Mario. 1990. *Typesetting BSI VDM with LaTeX.* Department of Computer Science, University of Manchester.

Woodcock, J., & Loomes, M. 1988. *Software Engineering Mathematics.* Pitman.