Z BASE STANDARD
VERSION 1.0

by

S. M. Brien
J. E. Nicholls

Technical Monograph PRG-107
ISBN 0-902928-84-8

November 1992

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Electronic mail: `Stephen.Brien@comlab.ox.ac.uk`

Acknowledgement

Preparation of this paper has been undertaken as part of the ZIP project.

ZIP — A unification initiative for Z Standards, Methods and Tools has been partially funded by the Department of Trade and Industry and the Science and Engineering Council under their joint Information Engineering Advanced Technology Programme.

Contributors

The Z notation and its mathematical foundations have been developed by many people. A selected list of papers tracing a history of Z development is included in the *References* at the end of this document (see page 201).

In addition to the listed contributors to the mathematical foundations of Z, many programmers, systems designers and architects have provided support by using Z and giving feedback to the designers of the notation.

Z Base Standard. This version of the Z Base Standard has been written and edited as follows:

Editors:	Stephen Brien
	John Nicholls
Chapter authors:	Stephen Brien
	Trevor King
	John Nicholls
	Jim Woodcock
	John Wordsworth
Additional contributions by:	Paul Gardiner
	Steve King
	Peter Lupton
	Ib Sørensen
	Rob Arthan
	Roger Jones
	Chris Sennett
	...

A more complete acknowledgement of contributions to the development of Z will be included in a *History of Z* to be published separately.

Z Standards Review Committee

Development of the Z Base Standard as part of the ZIP project has benefited from contributions and reviews by the *Z Standards Review Committee*. The following list includes members of the SRC and their alternates or deputies.

Derek Andrews	University of Leicester UK
Mark Ardis	AT & T Bell Laboratories Naperville USA
Stephen Brien	Oxford University PRG UK
Elspeth Cusack	British Telecom UK
Ray Crispin	Hewlett Packard Laboratories Bristol UK
John Dawes	ICL Winnersb Wokingham UK
Jim Farr	CESG Cheltenham UK
Susan Gerhart	AFM Inc Texas USA
Will Harwood	Imperial Software Technology Cambridge UK
Ian Hayes	University of Queensland Australia
Pier Luigi Iachini	INTECS Pisa Italy
Randolph Johnson	CESG Cheltenham UK
Roger Jones	ICL Winnersh Wokingham UK
Steve King	Oxford University PRG UK
Trevor King	Praxis plc Bath UK *(SRC secretary)*
Peter Lupton	IBM United Kingdom Laboratories Hursley UK
John McDermid	York University & York Software Engineering UK
Silvio Meira	University of Pernambuco Brazil
John Nicholls	Oxford University PRG UK *(Chair)*
Colin Parker	British Aerospace Warton UK
Jan Peleska	DST Kiel Germany
Ben Potter	BNR Europe Harlow UK
Brian Ritchie	Rutherford Appleton Laboratories Chilton UK
Gordon Rose	University of Queensland Australia
Mayer Schwartz	Tektronix Oregon USA
Chris Sennett	DRA Malvern UK
Ib Sørensen	BP Research Sunbury UK
Susan Stepney	Logica Cambridge UK
Bernard Sufrin	Oxford University PRG UK
Kees van Hee	Waterloo University Ontario Canada
Jeremy Wilson	British Telecom UK
Jim Woodcock	Oxford University PRG UK
John Wordsworth	IBM United Kingdom Laboratories Hursley UK

Members of the Standards Review Committee have been invited to join the Standards Panel: *IST/5/-/52: Z Notation* currently being formed.

Contents

Foreword

This is the current version of a Base Standard for the Z notation and is distributed for review and comment. This version has been specifically prepared for distribution at the Seventh **Z User Meeting** in London on 14th-15th December 1992, and will be made available for general distribution after that date.

The Z Base Standard is subject to change during its review by the Z Standards Review Committee and the BSI Standards Panel now being formed. New versions will be issued as needed.

Comments on this version of the Z Base Standard are welcomed and should be sent to

Editors Z Base Standard
c/o Secretary Z Standards Project
Oxford University Computing Laboratory
Programming Research Group
11 Keble Road, Oxford OX1 3QD
United Kingdom

The Z Base Standard has been produced as part of the work of the Z standards project, part of the ZIP project (IED project No. 1639).

0 Introduction

Z was originally developed as a *specification* notation for preparing formal descriptions of systems, without necessarily indicating how they will be implemented. This section includes a description of the aims and objectives of formal specification notations, with special reference to Z. The design principles used in the development of the Z standard are described.

0.1 Notations for system description

It is widely acknowledged that natural languages and similar informal notations have many disadvantages when used for writing technical descriptions. In using such languages it is difficult to write specifications with the required precision, clarity and economy of expression and to transform them systematically and reliably into code or hardware. Furthermore, it is impossible to carry out formal mathematical reasoning about informally written descriptions.

In contrast, specifications written in *formal* notations can be made precise and clear. Inference rules derived from their mathematical foundations enable designers to carry out mathematical reasoning and construct proofs relating to the properties of system descriptions.

The advantages of formal notations were recognised from an early stage in the history of computing, although it has taken considerable time for their practical application to become established. Many of the early large-scale applications of formal notation were for the specification of programming languages; formal descriptions of syntax are now widespread and for some languages there are formal descriptions of semantics.

Formal notations are now being used in a wide and expanding variety of environments, especially in key areas where the integrity of systems is critical, or where there is high intensity of use. For a discussion of domains of application for formal methods, see [16].

Examples of the effective use of formal specification notations are found in the following areas:

safety critical systems
security systems
the definition of standards
hardware development
operating systems
transaction processing systems

Descriptions of case studies from these and other application areas for Z are listed in a *Z Bibliography* by Bowen [2].

0.2 Objectives of a specification notation

The objectives of a formal specification notation are to assist in the production of descriptions that are complete, consistent and unambiguous. To achieve these objectives, a formal specification notation needs to be:

usable by those who read and write formal documents;

expressive, so that it can be used for a wide range of applications;

precise, so that it is possible to write descriptions that mean exactly what is intended;

given a *mathematically sound* meaning, since mathematical reasoning may be used in the development process;

suitable for defining sufficiently *abstract* models of systems that specifications do not need to contain unnecessary implementation details.

0.3 Characteristics of Z

A central part of Z is taken from the mathematics of set theory and first order predicate calculus. For the purposes of system description additions have been made to conventional mathematics, including:

a *type system* which requires each variable to be associated with a declared type. The ability to type-check a specification helps in assuring that it is accurate and consistent;

the *Z schema notation*, which provides a technique for grouping together and re-using common forms;

a *deductive system* which supports reasoning about Z specifications.

In addition, the following have been developed to help in the pragmatic use of Z in development projects:

the capability for writing explanatory text as an integral part of a Z document.

the inclusion within the standard of an agreed method of representing text in computers and transmitting it.

0.4 Design principles

The following design principles have been used in the development of the standard and are based on those used, explicitly or implicitly, in the original design of Z.

Basis in mathematics. Z is based on a central core of mathematics and uses accepted mathematical concepts and notation. In addition, there are means of defining and checking the *types* of Z elements and, by means of the Z *schema*, for structuring specifications.

Utility. All parts of Z included in the standard will have been shown to contribute to the main objectives of Z and will have beeu used in significant case studies or development projects.

Simplicity. There is an objective to keep the Z notation as simple as possihle, consistent with its overall objectives.

0.5 Aims of standardisation

The Z standard supports the following general aims of standardisation as listed in the British Standards Institution *Standard for Standards* [4]:

provision of a medium for communication and interchangeability;

support for the economic production of standardised products and services;

the establishment of means for ensuring consistent quality and fitness for purpose of goods and services;

promotion of international trade.

0.6 Validation of the standard

In order to *validate* the standard, it is necessary to ensure that it is is appropriate, consistent and complete, and is in accordance with the general understanding of the Z notation. In order to achieve this, the following steps have been taken:

existing descriptions of the notation have been used as a basis for the document;

alternative concepts and notations have been proposed where existing ones were considered deficient;

the standard is being reviewed by the *Z Standards Review Committee*, which includes experts in formal methods, users and tool makers;

the standard is being reviewed by the ZIP tools project to confirm that it can be supported by tools;

the mathematical part of the standard is being checked for soundness.

0.7 History of Z

This section (in preparation) will include a list of selected design papers on Z will identify some of the key decisions made during its development.

1 Scope and conformance

1.1 Scope of the Z Standard

The Z standard defines the representation, structure and meaning of the formal part of specifications written in the Z notation.

In addition to defining the formal part of the Z notation, the Z standard defines:

a Library or Toolkit of mathematical functions for use in writing Z specifications;

an Interchange Format for Z documents that enables them to be prepared, stored and transmitted within computer networks;

a deductive system for formal reasoning about Z specifications.

A Z document may contain both formal and informal text. The lexis of the standard does not define how the formal and informal parts are delimited; this is defined in the Interchange Format. The Interchange Format does not define the structure of the informal part of a Z document.

The standard does not define a method of using Z.

1.2 Conformance

A specification conforms to the standard for the Z notation if and only if the formal text is written in accordance with the syntax rules and is well typed.

A deductive system for Z conforms to the standard if and only if its rules are sound with respect to the semantics.

2 Semantic Metalanguage

In the following sections we describe the metalanguage used for defining the semantics of Z. We include:

- the names of all metalanguage symbols;
- the forms in which they are used;
- descriptions of their meaning.

Many of the symbols used in the semantic metalanguage are derived from conventional mathematics and are defined informally. Throughout the standard, the mathematical treatment is based on the Zermelo-Fraenkel (ZF) axiomatisation of set theory. An introduction to ZF theory can be found in text books on set theory—see for example Enderton [6] or Hamilton [9].

In addition to conventional mathematical symbols, we introduce and define a number of special symbols which allow concise semantic definitions to be written. Where these are similar to the symbols of Z, Z-like symbols are used and the following additional information is given:

- definitions of new symbols in terms of basic symbols. (or other new symbols)

Note that, although symbols similar to those of Z are used, the semantic metalanguage is not Z but standard mathematics, based on classical set theory.

Naming conventions. The following naming conventions are used:

upper-case letters $A, B, C, \ldots$ are used for sets;

lower-case letters $x, y, z, \ldots$ are used for elements of sets.

Commuting diagrams. In several of the following descriptions *commuting diagrams* are used to illustrate relationships between the set constructors being defined. Commuting diagrams are graphs whose nodes are labelled with sets. Nodes are connected by arrows, each arrow being labelled with a relation between the sets at each end. A diagram is said to *commute* when the composition of two different routes between nodes yields the same result.

2 SEMANTIC METALANGUAGE

2.1 Definitions and declarations

Variables and notations are introduced and named as follows:

Table 1: Declarations and definitions

Name	Symbol	Example	Description
declaration	:	$A : B$	A is declared to be an element of the set B
definition	$\widehat{=}$	$A \widehat{=} B$	A is defined as B

2.2 Sets

The following sets are predefined:

Table 2: Predefined sets

Name	Form	Description
empty set	$\varnothing$	the set having no elements.
integers	$\mathbb{Z}$	$\ldots, -2, -1, 0, 1, 2, \ldots$
strings	$\mathbb{S}$	the set of all strings of characters.

Relationships between sets and their members are written as follows:

Table 3: Relationships between sets and members

Name	Form	Description
membership	$x \in A$	x is a member of A.
subset	$A \subseteq B$	A is a subset of B i.e. all elements of A are elements of B.
equality	$A = B$	sets A and B are equal i.e. A and B have the same members.

2.2.1 Set constructors

The following ***set constructors*** define sets constructed from elements or from other sets:

Table 4: Set constructors

Name	Form	Description
set extension	$\{a, b, c, \ldots\}$	the set comprising elements $a, b, c, \ldots$
union	$A \cup B$	the set comprising all the elements of A and all the elements of B.
generalised union	$\bigcup A$	the set comprising all the elements of each set in A.
intersection	$A \cap B$	the set comprising the elements common to A and B.
set difference	$A \setminus B$	the set comprising the elements of A that are not elements of B.
power set	$\mathbf{P}\, A$	the set of all subsets of A.
finite power set	$\mathbf{F}\, A$	the set of all finite subsets of A.

2.3 Tuples and products

The following *constructors* define tuples and products:

Table 5: Tuples and products

Name	Form	Definition
tuple	$< x_1, \ldots, x_n >$	ordered list of the elements $x_1, \ldots, x_n$.
tuple projection	π_i	the *ith* member of a tuple. $\pi_i < x_1, \ldots, x_i, \ldots x_n > = x_i$ where $1 \leq i \leq n$
Cartesian product	$A_1 \times \ldots \times A_n$	the set of tuples $< x_1, \ldots, x_n >$ such that $x_1 \in A_1$ and ... and $x_n \in A_n$
enumerated product	A^i	the set of tuples $< x_1, \ldots, x_i >$ such that $x_1, \ldots, x_i \in A$
generalised product	A^+	$A^1 \cup A^2 \cup A^3 \cup \ldots$

2.4 Relations

In the following table, R, S denote binary relations, A and B denote sets.

Tahle 6: Relations

Name	Form	Definition
binary relations	$A \leftrightarrow B$	$\mathsf{P}(A \times B)$
identity relation	1_A	$< x, y > \in 1_A \Leftrightarrow x = y \wedge x \in A$
domain	$\mathrm{dom}\, R$	$x \in \mathrm{dom}\, R \Leftrightarrow \exists\, y \bullet < x, y > \in R$
range	$\mathrm{ran}\, R$	$y \in \mathrm{ran}\, R \Leftrightarrow \exists\, x \bullet < x, y > \in R$
converse	R^{-1}	$< x, y > \in R^{-1} \Leftrightarrow < y, x > \in R$
backward composition	$R \circ S$	$< x, y > \in R \circ S$
		$\Leftrightarrow \exists\, z \bullet < x, z > \in S < z, y > \in R$
forward composition	$R\, ;\, S$	$S \circ R$
range restriction	$R \rhd A$	$R\, ;\, 1_A$
domain restriction	$A \lhd R$	$1_A\, ;\, R$
relational override	$R \oplus S$	$((\mathrm{dom}\, R - \mathrm{dom}\, S) \lhd R) \cup S$
relational image	$^{\exists}(R)A$	$\mathrm{ran}(R \circ 1_A)$

2.5 Set constructors as relations

Set constructors can be given relational equivalences. By explicitly defining the domain of each constructor an equivalent set-theoretic relation can be constructed.

Table 7: Set constructors defined as relations

Name	Symbol	Domain	Range	Definition
union	$\cup$	$X \times X$	$\mathbf{P} \bigcup X$	$<< x_1, x_2 >, y > \in (\cup) \;\Leftrightarrow\; y = x_1 \cup x_2$
intersection	$\cap$	$X \times X$	$\mathbf{P} \bigcup X$	$<< x_1, x_2 >, y > \in (\cap) \;\Leftrightarrow\; y = x_1 \cap x_2$
subset	$\supseteq$	X	$\mathbf{P} \bigcup X$	$< x, y > \in (\supseteq) \;\Leftrightarrow\; y \subseteq x$
element	$\ni$	X	$\bigcup X$	$< x, y > \in (\ni) \;\Leftrightarrow\; y \in x$
singleton	$\{-\}$	X	$\mathbf{P}\, X$	$< x, y > \in \{-\} \;\Leftrightarrow\; y = \{x\}$
power	$\mathbf{P}$	X	$\mathbf{P}\,\mathbf{P} \bigcup X$	$< x, y > \in \mathbf{P} \;\Leftrightarrow\; y = \mathbf{P}\, x$
relational image	$^{\exists}(R)$	$\mathbf{P}$ dom R	$\mathbf{P}$ ran R	$< x, y > \in {}^{\exists}(R) \;\Leftrightarrow\; y = {}^{\exists}(R) x$
singleton image	$^{\wedge}(R)$	dom R	$\mathbf{P}$ ran R	$< x, y > \in {}^{\wedge}(R) \;\Leftrightarrow\; y = {}^{\exists}(R)\{x\}$
projection	π_i	$X_1 \times \ldots \times X_n$	X_i	$<< x_1, \ldots, x_n >, y > \in (\pi_i)$ $\Leftrightarrow y = x_i$
Cartesian product	X	X^+	$\mathbf{P}((\bigcup X)^+)$	$<< x_1, \ldots, x_n >, y > \in (\mathsf{X})$ $\Leftrightarrow y = x_1 \times \ldots \times x_n$

These relations will be used only when they have well-defined domains.

The following diagram shows commuting properties of relational constructors:

$$\begin{tikzcd}
\mathbf{P}\, A \arrow[r, "^{\exists}(R)"] & \mathbf{P}\, B \arrow[d, "\ni"] \\
A \arrow[u, "\{-\}"] \arrow[ur, "^{\wedge}(R)"] \arrow[r, "R"'] & B
\end{tikzcd}$$

2.6 Functions

A function is a relation with the property that for each element in its domain there is exactly one corresponding element in its range.

Table 8: Functions

Name	Form	Description or defiuition
total functions	$A \rightarrow B$	the set of functions from A into B whose domains are A. A total function is a function whose domain is its source.
total injections	$A \rightarrowtail B$	the set of total functions from A into B which are one-to-one.
total snrjections	$A \twoheadrightarrow B$	the set of total functions from A into B whose ranges are B.
bijections	$A \rightarrowtail\!\!\!\!\rightarrow B$	$A \rightarrowtail B \cap A \twoheadrightarrow B$
partial functions	$A \nrightarrow B$	${}^{3}(\supseteq)(A \rightarrow B)$
finite fnnctions	$A \nrightarrow\!\!\!\!\nrightarrow B$	$A \nrightarrow B \cap \mathbf{F}(A \times B)$
constant function	x_A°	maps all elements in the set A to x

In the remainder of this section, the term *function*, when not otherwise specified, is taken to mean *partial* fnnction.

Compatible functions. Two functions are said to be *compatible* if their union is a function.

The set of pairs of compatible functions from A to B is defined as follows:

$$C_{AB} = \mathrm{dom}(\cup \rhd A \twoheadrightarrow B)$$

The functional forms of the set operators: *union*, *intersection* and *set difference* are defined only when the arguments are compatible functions. When defined, they have the same value as their set equivalents.

Table 9: Compatible functions

Name	Symbol	Definition
Functional union	$\sqcup_{AB}$	$C_{AB} \lhd (\cup)$
Functional intersection	$\sqcap_{AB}$	$C_{AB} \lhd (\cap)$
Functional difference	$-_{AB}$	$C_{AB} \lhd (\setminus)$

2.7 Tuple and product constructors

The following tuple and product constructors are used.

The relational tuple $\langle R_1, \dots, R_n \rangle$ is a relation from the common domain of $R_1, \dots, R_n$ to the Cartesian product of their ranges.

The relational product $R_1 \times \dots \times R_n$ is a relation from the Cartesian product of the domains of $R_1, \dots R_n$ to the cartesian product of their ranges.

Table 10: Tuple and product constructors

Name	Form	Definition
relational tuple	$\langle R_1, \dots, R_n \rangle$	$< x, < y_1, \dots, y_n >> \in \langle R_1, \dots R_n \rangle$ $\Leftrightarrow < x, y_1 > \in R_1 \wedge \dots \wedge < x, y_n > \in R_n$
relational product	$R_1 \times \dots \times R_n$	$\langle \pi_1 ; R_1, \dots, \pi_n ; R_n \rangle$
general relational product	R^+	$R \cup (R \times R) \cup (R \times R \times R) \cup \dots$

The following diagram shows relationships between these constructors:

$$\begin{tikzcd}
 & B_i \\
B_1 \times \dots \times B_n \arrow[ur, "\pi_i"] & \\
 & A_1 \cap \dots \cap A_n \arrow[ul, "\langle R_1, \dots, R_n \rangle", "\subseteq"'] \arrow[uu, dash, "R_i"']
\end{tikzcd}$$

Theorem 2.1 *Relational tupling distributes through intersection as follows:*

$$\vdash (R_1, \ldots, R_n) \cap (S_1, \ldots, S_n) = (R_1 \cap S_1, \ldots, R_n \cap S_n)$$

The following diagram illustrates the properties of the product constructor:

$$\begin{tikzcd}
B_1 \times \ldots \times B_n \arrow[r, "\pi_i"] & B_i \\
A_1 \times \ldots \times A_n \arrow[u, "R_1 \times \ldots \times R_n"] \arrow[r, "\pi_i"'] & A_i \arrow[u, "R_i"']
\end{tikzcd}$$

Theorem 2.2 *Product distributes through intersection as follows:*

$$\vdash R_1 \times \ldots \times R_n \cap S_1 \times \ldots \times S_n = R_1 \cap S_1 \times \ldots \times R_n \cap S_n$$

2.8 Promoted application

Promoted application ($R \bullet S$) is the relational analogue of the S combinator in combinatory logic.

> **Note:** Promoted application is defined so that the following equality holds:
>
> $$(R \bullet S)_\rho = (R_\rho)_{(S_\rho)}$$
>
> where R_ρ is the application of the function R to the argument ρ.

Definition 2.1 *The* promoted application *operator constructs a relation from two other relations. Its effect is to apply the result of R to the result of S:*

$$R \bullet S \mathrel{\widehat{=}} (R \mathbin{;} \ni \cap S \mathbin{;} \pi_1^{-1}) \mathbin{;} \pi_2.$$

> **Note:** If R and S are functions and ρ is in both of their domains, then the tuple $(\rho, (p, q))$ belongs to the first part of this composite relation providing that that (p, q) is a member of the set R_ρ and p is S_ρ. The tuple (ρ, q) belongs to the composite relation exactly when for some p the tuple $(\rho, (p, q))$ belongs to the first part.

Promoted application is disjunctive in both arguments.

A derived form of promoted application is the apply-to-n function: $(\_n)$.

Definition 2.2 *The apply-to-n function takes as an argument a function and has as a result the application of that function to the element n. It is defined as follows:*

$$(\_n) \mathrel{\widehat{=}} (1 \bullet n^\circ)$$

> **Note:** If ρ is a function and n is an element of the domain of ρ then the following equality holds:
>
> $$(\_n)_\rho = \rho_n \,.$$

3 Semantic Universe

This section defines a semantic universe within which the meanings of Z specifications lie; it is based on the Zermelo-Fraenkel axiomatisation of sets discussed in the last section. It contains the meanings of names, types, and values used in a specification, as well as the environment used to define the overall meaning of a specification.

3.1 Names and Types

Our first task in building our universe is to explain the use of names and the notion of types. In Z, a name is used to denote an element, which may be a set, a tuple, a binding, or an element of a given type. These names come in three varieties: they may be the names of schemas, variables, or constants. This partitioning of names is dependent on the specification in question, the members of each set not being distinguishable in the concrete syntax. Abstractly, we have that our set of names *Name* is partitioned into schema names, variable names, and constant names:

$$\langle \mathit{SchemaName}, \mathit{Variable}, \mathit{Constant} \rangle \text{ partition } \mathit{Name}.$$

In common with other specification and programming languages, but unlike ZF set theory, the rules of Z require that every name introduced in a Z specification is given a particular type which determines the possibilities for the values that it may take. This type has several purposes, both practical and technical. It offers the usual advantages with which we are familiar in programming languages: it helps to make the specification easier to understand, and it permits a certain mechanical checking of a specification to be done. It also guarantees that Russell's paradox is avoided in a specification, and that sets defined in comprehension exist. Finally, it provides an insulation against the details of the encoding of Z constructs in ZF set theory.

The simplest types are ***given set names***, which are used to introduce abstract objects into a specification, or as the formal names of generic parameters. Their names are drawn from the set *Constant*.

$$\mathit{GivenSetName} \subseteq \mathit{Constant}$$

Note: The names for the set of integers $\mathbb{Z}$ and the set of strings $\mathbb{S}$ are members of the set of given set names.

For more complicated types, Z provides three type constructors so that power set types, Cartesian product types, and schema types may be introduced. If $n_1, \ldots, n_m$ are names, and $\tau_1, \ldots, \tau_m$ represent types, then the following all represent types:

$$\mathbb{P}\,\tau_1,$$
$$\tau_1 \times \ldots \times \tau_m,$$
$$[\; n_1 : \tau_1 ; \ldots n_m : \tau_m \;].$$

Every type belongs to the semantic set *Type*, which is partitioned into the four subsets *Gtype*, *Ptype*, *Ctype*, and *Stype* representing the given types, power set types, Cartesian product types, and schema types:

$$\langle \mathit{Gtype}, \mathit{Ptype}, \mathit{Ctype}, \mathit{Stype} \rangle \text{ partition } \mathit{Type}.$$

It is easy to think of something of given type as an object, of power set type as a set, of Cartesian product as a tuple, but what about something of schema type? As we can see from the above example, it is a function from variable names to types; such a function is called a signature:

$Signature \mathrel{\widehat{=}} Variable \nrightarrow Type.$

Now we have everything that we need in order to explain the structure of the set of types. Consider power set types. From every type τ, we can construct the unique type which is $\mathbb{P}\,\tau$; every power set type $\mathbb{P}\,\tau$ is constructed in this way from a unique type τ. Thus, the power set constructor is a *bijection* between *Type* and *Ptype*. Similar arguments apply to the other type constructors. We can sum this up by defining the following four bijections with the partitions of *Type*:

$givenT : GivenSetName \rightarrowtail Gtype$
$powerT : Type \twoheadrightarrow Ptype$
$cproductT : Type^{+} \twoheadrightarrow Ctype$
$schemaT : Signature \twoheadrightarrow Stype.$

For each specification there is a set of distinct given types. All other types used are constructed from these given types using a *unique* combination of the type constructors. This uniqueness is guaranteed because the type constructors are in bijection with the partitions of the set *Type*. Therefore the set *Type* is the smallest set which is closed under these type constructors. *Type* is the initial algebra over the signature given by $givenT$, $powerT$, $cproductT$, $schemaT$.

3.2 Values in Z

As we said above, one of the purposes of ascribing a type to a variable is to determine which values the variable may take. To make this possible, each type has a set of values associated with it, called its *carrier set*. The values in the carrier set of a given type are regarded as atomic objects. Each value in the carrier set of a non given type is modelled by a ZF set. The relationship between the types and values in a specification is defined by the function *Carrier*, whose definition we approach inductively.

> **Note:** In Z a type is identified by its carrier set. In the previous examples τ was the carrier set for some type.

Definition 3.1 *For each specification there is a carrier function which maps the given types to elements of W_0.*

$Carrier_0 : Gtype \rightarrow W_0$

Now, suppose that τ is a given type; what is the carrier set of the power set type $\mathbb{P}\,\tau$? It is simply the set $\mathbb{P}(Carrier\ \tau)$. In general, for a power set type σ, we must calculate the carrier set by stripping off the power set constructor, calculating the carrier set of this underlying type, and then forming the power set of the result; formally, this is given by the expression

$(powerT^{-1} \mathbin{;} Carrier_0 \mathbin{;} \mathbb{P})\ \sigma.$

Similarly, if σ is a Cartesian product of given types, then we should break it up into its constituent given types, work out their carrier sets, and then form their Cartesian product, so that we end up with a set of tuple values:

$$(\mathit{cproductT}^{-1} \mathbin{;} \mathit{Carrier}_0^{\dagger} \mathbin{;} \times)\ \sigma.$$

Finally, if σ is a schema type made out of given types, then we should obtain the underlying signature; this yields a function from names to types, which we must turn into a function from names to the carrier sets of these types; finally, we must form the schema product, so that we end up with a set of functions from names to values:

$$(\mathit{schemaT}^{-1} \mathbin{;} {}^{\exists}(1 \times \mathit{Carrier}_0) \mathbin{;} \times_{\mathit{Name}})\ \sigma.$$

In this discussion, we have been assuming that the type constructors are applied to given types, but in general they are applied to arbitrary types. Since a type is made out of a finite sequence of applications of the constructors, we can define the *depth* of a type to be the length of this sequence. Now we can give our inductive definition using this notion of depth:

Definition 3.2

$$\begin{aligned}
&\mathit{Carrier}_{i+1} \mathrel{\widehat{=}} \\
&\quad \mathit{Carrier}_i \\
&\quad \cup\ \mathit{powerT}^{-1} \mathbin{;} \mathit{Carrier}_i \mathbin{;} \mathbb{P} \\
&\quad \cup\ \mathit{cproductT}^{-1} \mathbin{;} \mathit{Carrier}_i^{\dagger} \mathbin{;} \times \\
&\quad \cup\ \mathit{schemaT}^{-1} \mathbin{;} {}^{\exists}(1 \times \mathit{Carrier}_i) \mathbin{;} \times_{\mathit{Name}}.
\end{aligned}$$

In order to calculate the carrier set for a type τ, we must apply $\mathit{Carrier}_i$, where i is the depth of type τ. Notice that every carrier function whose domain contains τ gives the same result for τ: this justifies our general definition.

Definition 3.3 *The general carrier function which maps elements of Type to their carrier sets is defined as follows:*

$$\mathit{Carrier} \mathrel{\widehat{=}} \mathit{Carrier}_0 \cup \mathit{Carrier}_1 \cup \mathit{Carrier}_2 \cup \ldots.$$

The values which may be used in a Z specification are those that are in the carrier sets that are assigned to the types. This set is constructed from the elements of W_0 using the type constructors.

Definition 3.4 *The set W of all values is the union of all the carrier sets for the elements of Type:*

$$W \mathrel{\widehat{=}} \bigcup \operatorname{ran} \mathit{Carrier}.$$

Definition 3.5 *A binding is a finite mapping from variables to values:*

$$\mathit{Binding} \mathrel{\widehat{=}} \mathit{Variable} \twoheadrightarrow W.$$

The carrier function is a homomorphism between types and W. Thus, we have the equations

$$\begin{aligned}
&\mathit{Carrier}(\mathit{powerT}\,\tau) = \mathbb{P}(\mathit{Carrier}\ \tau) \\
&\mathit{Carrier}(\mathit{cproductT}(\tau_1, \tau_2)) = (\mathit{Carrier}\ \tau_1) \times (\mathit{Carrier}\ \tau_2) \\
&\mathit{Carrier}(\mathit{schemaT}\ \sigma) = \times_{\mathit{Name}}({}^{\exists}(1 \times \mathit{Carrier})\sigma).
\end{aligned}$$

This is depicted in the commuting diagrams in figure 1.

$$\begin{tikzcd}
 & & & \mathit{Type+} \arrow[dl, "\mathit{Carrier+}"'] \arrow[dr, "\mathit{eproductT}"] & & & \\
 & & \mathit{W+} \arrow[ddr, "\mathbf{X}"'] & & \mathit{Ctype} \arrow[ddl, "\mathit{Carrier}"] & & \\
 & \mathit{W} \arrow[drr, "\mathbf{P}"] & & & & \mathit{Binding} \arrow[dll, "\mathbf{X}"'] & \\
\mathit{Type} \arrow[ur, "\mathit{Carrier}"] \arrow[dr, "\mathit{powerT}"'] & & & \mathit{W} & & & \mathit{Signature} \arrow[ul, "\mathbf{E}_{(I \times \mathit{Carrier})}"'] \arrow[dl, "\mathit{schemaT}"] \\
 & \mathit{Ptype} \arrow[urr, "\mathit{Carrier}"] & & & & \mathit{Stype} \arrow[ull, "\mathit{Carrier}"'] & \\
 & & & \mathit{Gtype} \arrow[uu, "\mathit{Carrier}"'] & & & \\
 & & & \mathit{Constant} \arrow[u, "\mathit{givenT}"'] & & &
\end{tikzcd}$$

Figure 1: The type system.

3.3 Elements in Z

Each element in Z is represented by the pair consisting of its type and its value. The semantic set *Elm* is a set of type-value pairs; this set may be considered as the relation between types and values in which a type is related to a value if and only if the value is a member of the carrier set of the type.

Definition 3.6 *A value is an element of a type if and only if it is contained in the carrier set of the type:*

$$Elm \mathrel{\widehat{=}} Carrier \mathbin{;} \ni .$$

The first and second projections on a tuple are used to extract the type and value respectively.

Definition 3.7 *The type and value functions are projections from the tuples in Elm:*

$$t \mathrel{\widehat{=}} \pi_{1Elm},$$
$$v \mathrel{\widehat{=}} \pi_{2Elm}.$$

The type function is a surjection since the carrier set of each type is non-empty. Since the carrier set of each type contains at least one value, *Elm* contains at least one pair for each type; thus, t is *surjective*. Since the values that may be used in a specification are all to be found in the carrier sets, *Elm* contains at least one pair for each value; thus, v is *surjective*.

Definition 3.8 *The membership relation for elements* $\exists$ *is the lifted form of a type value pair:*

$$\exists \mathrel{\widehat{=}} (powerT^{-1} \times \ni)$$

Suppose that we have a Z specification. It consists of a number of definitions which introduce names. Each name may denote some value, and each name must have some type: that is, each name may be associated with an *element*. We call such an assignment of elements to names a *situation*.

Definition 3.9 *A situation is a finite mapping from variables to elements:*

$$Situation \mathrel{\widehat{=}} Variable \twoheadrightarrow Elm.$$

A situation tells us two things about the names in a specification: their types and their values. If we think about the type projection of each name, then we obtain a mapping from names to types: a signature. If, on the other hand, we think about the value projection of each name, then we obtain a mapping from names to values: a binding. The signature and binding corresponding to a particular situation can be extracted by the functions T and V respectively.

Definition 3.10 *The T and V functions are defined as follows:*

$$T \mathrel{\widehat{=}} {}^{\exists}(1 \times t)$$
$$V \mathrel{\widehat{=}} {}^{\exists}(1 \times v).$$

The following commuting diagram illustrates the relationship between types and values and their lifted forms as signatures and bindings:

$$\begin{tikzcd}
& \mathbb{P}\,\mathit{Variable} & \\
\mathit{Signature} \arrow[ur, "\mathrm{dom}"] \arrow[d, "\mathrm{ran}"'] & \mathit{Situation} \arrow[l, "T"'] \arrow[r, "V"] \arrow[u, "\mathrm{dom}"] \arrow[d, "\mathrm{ran}"] & \mathit{Binding} \arrow[ul, "\mathrm{dom}"'] \arrow[d, "\mathrm{ran}"] \\
\mathbb{P}\,\mathit{Type} & \mathbb{P}\,\mathit{Elm} \arrow[l, "{}^{3}(t)"] \arrow[r, "{}^{3}(v)"'] & \mathbb{P}\,W
\end{tikzcd}$$

Since the product constructor and the image constructor preserve surjectivity, T is a surjective function. Our next theorem follows from this.

Theorem 3.1 *The type of the set of situations is exactly the set of signatures:*

$$\vdash {}^{3}(T)\mathit{Situation} = \mathit{Signature}.$$

3.4 Generics

A Z expression that involves a generic instantiation acquires a type and a value that depends upon the type and value of the expression used in the instantiation. Thus if we see $\varnothing[\mathbb{N}]$ we know this has a different type from $\varnothing[\mathbb{P}\ \mathbb{N}]$. The various types that $\varnothing$ may take are represented as a *function* from type to type. In the case of $\varnothing$, this function takes an arbitrary powerset type to itself. In general, where a generic definition contains a list of identifiers, the various possible instantiations are a function from lists of elements to a type and value. The elements which may appear as actual parameters of a generic definition must be of powerset type.

3.4.1 Generic Types

For each generic type the number of formal parameters is fixed, and every possible sequence of powerset types with the right number of formal parameters is given a type. So each generic type is a function from fixed-length sequences of power types to a type.

Definition 3.11 *For any natural number $n > 0$, the set of all generic types with n parameters is defined as follows:*

$$Gen\_Type_n \mathrel{\widehat{=}} Ptype^n \rightarrow Type.$$

Definition 3.12 *The set of all generic types is the union of all the sets of fixed length generic types:*

$$Gen\_Type \mathrel{\widehat{=}} Type \cup Gen\_Type_1 \cup Gen\_Type_2 \cup \ldots.$$

If X and Y are generic formal parameters and a generic definition declares $x : X;\ y : Y$. Then an expression such as $x \in y$ or $x = y$ would impose a mutual constraint on the types that could be used to instantiate X and Y. For $x \in y$, we have the constraint that the types that Y may take are the powerset of the types that X may take; for $x = y$, we have the constraint that the types that Y may take must be the same as the types that X may take.

The definition of generic types as total functions imposes the constraint that generic definitions do not create inter-relationships between the type of their formal parameters. Such inter-relationships can always be eliminated within a specification.

Since all the type constructors are bijections, then any inter-relationship between the types of generic parameters is functional. Therefore any dependent parameters are redundant since they can be uniquely determined as functions of the other parameters. For $x \in y$ the inter-relationship can be eliminated by removing Y as a formal generic parameter and defining $y : \mathbb{P}\, X$; for $x = y$ we can eliminate Y and define $y : X$.

3.4.2 Generic Elements

As with generic types, for each generic element there is only one number of formal parameters that it can take; furthermore every possible sequence of the correct number of elements with powerset type is given a type and value.

Definition 3.13 *Generic elements are functions from tuples of set elements to elements:*

$$Gen\_Elm : \mathbb{P}(Pelm^+ \rightarrow Elm).$$

Sets in Z are those elements which have a power type:

Definition 3.14 *The set Pelm contains all elements which have power type:*

$$Pelm \mathrel{\widehat{=}} Ptype \lhd Elm.$$

The functions representing generic elements are type consistent: a generic element, when instantiated with two sequences of elements of the same type, will give two elements of the same type. In order to define this property it is necessary to characterise the type part of a generic element.

Definition 3.15 *The function τ takes a function from tuple of elements to elements and returns a generic type:*

$$\tau \mathrel{\widehat{=}} {}^{\exists}(I^+ \times I) \cup I$$

Definition 3.16 *All generic elements have a type part which is functional, i.e. contained in Gen_Type:*

$$Gen\_Elm \;\widehat{=}\; \mathrm{dom}(\tau \rhd Gen\_Type).$$

A theorem similar to that for elements holds for generic elements:

$$\vdash {}^{\exists}(\tau) Gen\_Elm \;=\; Gen\_Type.$$

3.5 Environments

In order to give a meaning to the constructs of Z, we need an environment to record the elements denoted by the names used in a Z specification.

Definition 3.17 *An environment is defined as a finite partial function from names to generic elements:*

$$Env \;\widehat{=}\; Name \twoheadrightarrow Gen\_Elm.$$

Whether a Z specification is well typed or not is a question that is independent of the *values* of the declared variables. To be able to answer this question it is necessary to have an environment in which the types of all names are recorded.

Definition 3.18 *A type-environment is defined as a finite function from names to generic types:*

$$Tenv \;\widehat{=}\; Name \twoheadrightarrow Gen\_Type.$$

The simple relationship between the richer environment, ENV, and the one used just for type checking, $TENV$, is given by the forgetful function Υ which throws away the values.

Definition 3.19 *The function Υ maps the second element of each tuple in an environment onto its corresponding generic type:*

$$\Upsilon \;\widehat{=}\; {}^{\exists}(1_{Name} \times \tau).$$

The following commuting diagram illustrates the relationship between the environment and type-environment:

$$\begin{tikzcd}
 & \mathbf{P}\,\mathit{Name} & \\
\mathit{Tenv} \arrow[ur, "\mathrm{dom}"] \arrow[dd, "\mathrm{ran}"'] & & \mathit{Env} \arrow[ul, "\mathrm{dom}"'] \arrow[ll, "\Upsilon"'] \arrow[dd, "\mathrm{ran}"] \\
 & & \\
\mathbf{P}\,\mathit{Gen\_Type} & & \mathit{power}\,\mathit{Gen\_Elm} \arrow[ll, "{}^{\exists}(\tau)"]
\end{tikzcd}$$

The function t used in the construction of Υ can be shown to be surjective onto *Type*, so the following theorem holds.

Theorem 3.2 *Every type environment has at least one corresponding full environment:*

$$\vdash {}^{\exists}(\Upsilon)\mathit{Env} \;=\; \mathit{Tenv}.$$

If T is a set of type environments, then ${}^{\exists}(\Upsilon^{-1})T$ is the corresponding set of meaning environments.

4 Language Description

This section provides an introdnction to the following sections by illustrating how the the syntax and semantics of Z are defined.

The following sections each define a major syntactic category: *expression*, *predicate*, *declaration*, *schema text*, *schema*, *paragraph*. Within each there are subsections corresponding to the syntactic categories of the abstract syntax. Each definition follows a consistent pattern and is sub-divided nnder the following headings: *Abstract Syntax*, *Representation and Transformation*, *Type*, and *Value/Meaning*. At the end of each section tables contain the definitions of the free variables of each element, together with their alphabet where appropriate. Finally a table of equivalences for substitntion is given.

A *denotational* style of semantic description is used [21] and, as in the customary style of writing denotational semantics, semantic brackets are used to delimit text for which denotations are given. The notation is extended by providing different shapes of brackets for different kinds of language elements as shown in the following table. Three types of semantic functions are used, for *type*, *value* and *meaning*. The different types are identified by superscripts on the hrackets.

Table 11: Semantic hrackets

Bracket	Argument	Forms
$\llbracket - \rrbracket$	Expression	$\llbracket - \rrbracket^{\mathcal{T}}, \llbracket - \rrbracket^{\mathcal{V}}, \llbracket - \rrbracket^{\mathcal{M}}$
$\{\!\vert - \vert\!\}$	Predicate	$\{\!\vert - \vert\!\}^{\mathcal{T}}, \{\!\vert - \vert\!\}^{\mathcal{V}}, \{\!\vert - \vert\!\}^{\mathcal{M}}$
$(\!\vert - \vert\!)$	Declaration	$(\!\vert - \vert\!)^{\mathcal{T}}, (\!\vert - \vert\!)^{\mathcal{M}}$
$(\!\!\vert - \vert\!\!)$	Schema	$(\!\!\vert - \vert\!\!)^{\mathcal{T}s}, (\!\!\vert - \vert\!\!)^{\mathcal{M}s}$
$\langle\!\vert - \vert\!\rangle$	SchemaText	$\langle\!\vert - \vert\!\rangle^{\mathcal{T}}, \langle\!\vert - \vert\!\rangle^{\mathcal{M}}$
$\langle\!\vert - \vert\!\rangle$	Paragraph	$\langle\!\vert - \vert\!\rangle^{\mathcal{T}}, \langle\!\vert - \vert\!\rangle^{\mathcal{M}}$

4 LANGUAGE DESCRIPTION

The following meta-variables will be used.

Variables	Sort
E, x, y	Expression
n, m	Name
a	String
i	Number
t	Tuple
s, u	Set
b	Binding
f	Function
P, Q	Predicate
C, D	Declaration
St	Schema Text
S, T	Schema
Par	Paragraph

4.1 Abstract syntax

For each language element, its abstract syntax is defined in a form of BNF. The following example illustrates the style used.

POWERSET = P EXP

In some cases symbols such as P are used rather than key-words or other structures in the syntax to make reading of the abstract syntax easier. The complete abstract syntax is presented in an Annex.

4.2 Representation and transformation

For each language element a table is provided showing the production or productions, expressed in the representation syntax, of the language element being defined and the relationship between the concrete and abstract forms.

Note: There may be more than one representation of an abstract syntax category; in such cases all forms are listed. In some cases the multiplicity of representations is due to the fact that some forms can be considered as abbreviations of others.

The transformation is presented in a denotational style with different superscripts on the brackets to denote the type of argument.

Table 12: Transformation Functions

Brackets	Argument
$[\![\_]\!]^{\mathcal{E}}$	Expression
$[\![\_]\!]^{\mathcal{P}}$	Predicate
$[\![\_]\!]^{\mathcal{D}}$	Declaration
$[\![\_]\!]^{\mathcal{S}}$	Schema
$[\![\_]\!]^{\mathcal{ST}}$	SchemaText
$[\![\_]\!]^{\mathcal{PAR}}$	Paragraph

The following example illustrates the tabular form in which the representation form is presented together with its transformation to its abstract form:

Production	Concrete	Abstract
'P' ,Expression5	P s	$P[\![s]\!]^{\mathcal{E}}$

In this example the production for power set shows how a power set is represented i.e. as an expression prefixed with the power set symbol. The second column is an example of this concrete form. In this case s is some expression. The third column gives the abstract form of this concrete expression. In this case the form is an (abstract) powerset symbol followed by the abstract form of the expression s. These two columns can be read as an equation in the form:

$$[\![\mathrm{P}\, s]\!]^{\mathcal{E}} \quad = \quad \mathrm{P}[\![s]\!]^{\mathcal{E}}.$$

The representation syntax is presented in a complete form in a later Annex.

4.3 Type

The definition of the Z type system is by structural induction over the abstract representation of a Z specification. The well-typedness of a Z specification can be determined independently of the *values* of the declared variables. So we see that the following definition of the Z type system is entirely self-contained: given a Z specification, the type definitions determine whether that specification is well-typed.

> **Note:** It is important to note that asking whether a certain specification is well-typed is decidable. Asking what the type of any term in a given environment is likewise decidable.

This is in marked contrast to evaluation, where asking whether a certain name may have a certain value is undecidable in general.

The fact that well-typing is decidable is not quite as obvious as all that, because *TENV* represents generic definitions using infinite objects. However, the infinite function from tuples of powerset type to type can always be represented as a finitary expression.

Name	Form	Sort
Expression Type	$[\![E\ ]\!]^{\mathcal{T}}$	$Tenv \nrightarrow Type$
Predicate Type	$\{\![P\ ]\!\}^{\mathcal{T}}$	$\mathbb{P}\ Tenv$
Declaration Type	$(\![D\ ]\!)^{\mathcal{T}}$	$Tenv \nrightarrow Signature$
Schema Type	$(\![S\ ]\!)^{\mathcal{T}s}$	$Tenv \nrightarrow Signature$
SchemaText Type	$(\!\{St\ \}\!)^{\mathcal{T}}$	$Tenv \nrightarrow Tenv$
Paragraph Type	$(\!\{Par\ \}\!)^{\mathcal{T}}$	$Tenv \nrightarrow Tenv$

The following example illustrates the description of the type of a powerset:

Type The type of the power set $\mathbb{P}\ s$ is the power set type of the type of the set s.

$$[\![\mathbb{P}\ s\ ]\!]^{\mathcal{T}} \quad = \quad ([\![s\ ]\!]^{\mathcal{T}} \rhd Ptype)\ ;\ powerT$$

Note: A power set $\mathbb{P}\ s$ is well typed only if s has power set type.

The type description contains an informal description, the mathematical definition of the type function for the powerset and an explanation of when it is well-typed. This last explanation is derived directly from the domain of the type function.

4.4 Meaning

The meanings of *expression, predicate, declaration, schema* and *paragraph* are given by the following functions.

Name	Form	Sort
Expression Meaning	$[\![E\ ]\!]^{\mathcal{M}}$	$Env \nrightarrow Elm$
Predicate Meaning	$\{\![P\ ]\!\}^{\mathcal{M}}$	$\mathbb{P}\ Env$
Declaration Meaning	$(\![D\ ]\!)^{\mathcal{M}}$	$Env \leftrightarrow Situation$
Schema Meaning	$(\![S\ ]\!)^{\mathcal{M}s}$	$Env \leftrightarrow Situation$
SchemaText Meaning	$(\!\{St\ \}\!)^{\mathcal{M}}$	$Env \leftrightarrow Env$
Paragraph Meaning	$(\!\{Par\ \}\!)^{\mathcal{M}}$	$Env \nrightarrow Env$

The meanings of *expression*, *predicate*, *declaration* and *schema* are combined to provide a meaning for a paragraph. This meaning is a relation between environments. The meaning of a specification is defined as the image of the empty environment through the composition of the paragraph relations.

The following example illustrates the description of the meaning of a simple declaration:

> **Meaning** The meaning of the simple declaration $n_1, \ldots, n_m : s$ is a relation from the environment to those situations which associate each of the names $n_1, \ldots, n_m$ with one of the elements of the set expression s:
>
> $$(\!| n_1, \ldots n_m : s |\!)^{\mathcal{M}} \;=\; [\![s]\!]^{\mathcal{M}} \,;\, \langle (n_1{}^{\circ}, \exists), \ldots, (n_m{}^{\circ}, \exists) \rangle \,;\, \{\ldots\}.$$
>
> **Note:** The simple declaration $n_1, \ldots, n_m : s$ is value-defined exactly when the expression s is a non-empty set.

The meaning description contains an informal description, the mathematical definition of the meaning function for the declaration and an explanantion of when it is value-defined. This last explanation is derived directly from the domain of the meaning fnnction.

4.5 Value

The meaning functions for expressions and predicates are defined in terms of their type and value. So the value functions are the primitives defined in the following sections. These functions have the following structure:

Name	Form	Sort
Expression Value	$[\![E]\!]^{\mathcal{V}}$	$Env \rightarrow W$
Predicate Value	$\{\!| P |\!\}^{\mathcal{V}}$	$\mathbb{P}\ Env$

The following example illustrates the description of the value of a powerset:

> The value of the power set $\mathbb{P}\, s$ is the set of all the subsets of the value of s:
>
> $$[\![\mathbb{P}\, s]\!]^{\mathcal{V}} \;=\; [\![s]\!]^{\mathcal{V}} \,;\, \mathbb{P}$$
>
> **Note:** A powerset $\mathbb{P}\, s$ is value-defiued only if the expression s is value-defined.

The value description contains an informal description, the mathematical definition of the value function for the powerset and an explanantion of when it is value-defined. This last explanation is derived directly from the domain of the value function.

4.6 Free variables.

Ordinarily the definition of the free variables of an expression can be considered as a fnnction on the names of identifiers appearing in the text of the expression and the variable bound by the declarations. In Z however, tbe case is somewhat more complicated. The use of schema references as declarations means that there is an implicit declaration. The names introduced by the declaration S where S is a schema reference are not related to the name S but to its value in the particular environment within which it is being evaluated. In other words the free variables of an expression depend on the text of the expression *and* the environment in which the expression is evaluated.

We define the free variables of an expression to be a partial function from environment to sets of names:

$$\phi_e(E) : Env \nrightarrow \mathbb{P}\, Name$$

The set of names defined as the free variables for an expression for a particular environment is the smallest set of names which must be in the environment in order for the expression to be well-defined. However since local declarations do not introduce schema references, the free variables of an expression are unchanged by a local declaration. So in the definitions we omit the environment parameter as it has no effect on the value of the free variables.

Table 13: Free Variable Function

Function	Argnment
ϕ_e	Expression
ϕ_p	Predicate
ϕ_d	Declaration
ϕ_s	Schema
ϕ_d	SchemaText

At the end of each section there is a table defining the free variable for each construct within that category. The following example illustrates the definition of the free variables of a power set:

Table 14: Extract from Table of Free Variables

Expression	Free Variables
...	
$\mathbf{P}\ x$	$\phi_e x$
...	

This can also be read as an equation in the following form:

$$\varphi_e \mathbf{P}\ s \quad = \quad \varphi_e s.$$

4.7 Alphabet

The syntactic categories of declaration, schema text and schema are used to introduce new names. These new names are called their alphabet. The alphabet is the set of the names in the signature as defined by the type rules (where applicable).

Table 15: Alphabet Function

Function	Argument
α	Declaration
	Schema
	SchemaText

Table 16: Extract from Table of Alphabets

Declaration	Alphabet
$n_1, \ldots, n_m : s$	$\{n_1, \ldots, n_m\}$
...	

This can also be read as an equation in the following form:

$$\alpha(n_1, \ldots, n_m : s) \quad = \quad \{n_1, \ldots, n_m\}.$$

4.8 Substitution

The tables of semantic equivalences for substituted expressions are given at the end of each section. These tables indicate when one expression can be replaced by another without changing the meaning.

The following example illustrates the semantic equivalence of substitution into a power set:

Table 17: Extract from Table of Semnantic Equivalences

Substitution	Equivalence
...	
$b \odot \mathbb{P}\, s$	$\mathbb{P}(b \odot s)$
...	

This can also be read as an equation in the following form:

$$b \odot \mathbb{P}\, s \quad \equiv \quad \mathbb{P}(b \odot s),$$

where the symbol $\equiv$ denotes semantic equivalence.

5 Expression

5.1 Introduction

As in computer languages, *expression* is a general form for defining values in Z.

In the abstract syntax given below, the different kinds of Z entity are listed. The entities included in the syntax, further defined in this chapter, may be subdivided as follows:

Elements:

IDENT GENINST NUMBERL STRINGL

These denote elementary values.

Set constructors:

SETEXTN SETCOMP POWERSET

These are used to construct sets from elements or sets

Tuple constructors:

TUPLE PRODUCT TUPLESELECTION

These are used to construct tuples from elements or tuples and select elements from tuples.

Binding constructors:

BINDINGEXTN THETAEXP SCHEMAEXP BINDSELECTION

These are used to construct bindings and select elements from bindings.

Functional forms:

FUNCTAPP DEFNDESCR

These represent function application and definite description.

Other Forms:

IFTHENELSE EXPSUBSTITUTION

These respectively represent a conditional expression and substituted expression.

Arithmetic and other expressions

In Z, facilities for defining arithmetic and string valued expressions such as those of programming languages are included in the *Z Toolkit*, where they are defined in terms of other Z constructions.

Abstract Syntax

```
EXP = IDENT
    | GENINST
    | NUMBERL
    | STRINGL
    | SETEXTN
    | SETCOMP
    | POWERSET
    | TUPLE
    | PRODUCT
    | TUPLESELECTION
    | BINDINGEXTN
    | THETAEXP
    | SCHEMAEXP
    | BINDSELECTION
    | FUNCTAPP
    | DEFNDESCR
    | IFTHENELSE
    | EXPSUBSTITUTION
```

Stages of definition

In this chapter definitions are built up in stages: first a *type function* is defined, then a *value function*. From these, a *meaning function* can be derived according to rules given below.

Type function For any expression E, its *type function* $[\![E]\!]^{\mathcal{T}}$ is a partial function from type-environments to types. The expression E is *well-typed* in exactly those type-environments contained in $\mathrm{dom}\,[\![E]\!]^{\mathcal{T}}$. The type of an expression in a type-environment is the result of applying its type function to that type-environment. The type function for an expression E is constructed from the type functions for its sub-expressions; thus the type of E is derived from the types of its sub-expressions.

The type of an expression in an environment is its type evaluated in the corresponding restricted type-environment. The function $\mathrm{T} \mathbin{;} [\![E]\!]^{\mathcal{T}}$ corresponds to the type function for E in the full meaning environment, where T is the function that restricts an environment to its corresponding type-environment. An expression is well-typed in an environment if and only if it is well-typed in the corresponding type environment.

Value function For any expression E, its *value function* $[\![E]\!]^{\mathcal{V}}$ is a partial function from environments to values. The expression E is *value-defined* in exactly those environments contained in $\mathrm{dom}\,[\![E]\!]^{\mathcal{V}}$. The value of an expression in a environment is the result of the application of its value function to that environment.

Meaning function From the type and value functions for an expression E it is possible to define a *meaning function* $[\![E]\!]^{\mathcal{M}}$. The meaning of an expression is the pair of its type and its value. The meaning function for an expression is constructed as follows:

$$[\![E]\!]^{\mathcal{M}} = (\Upsilon \,;\, [\![E]\!]^{\mathcal{T}} \,,\, [\![E]\!]^{\mathcal{V}})$$

The expression E is *well-defined* in exactly those environments contained in the set:

$$\mathrm{dom}(\Upsilon \,;\, [\![E]\!]^{\mathcal{T}} \,.\, [\![E]\!]^{\mathcal{V}})$$

This is equal to the set:

$$\mathrm{dom}\,\Upsilon \,;\, [\![E]\!]^{\mathcal{T}} \cap \mathrm{dom}\,[\![E]\!]^{\mathcal{V}}$$

Thus an expression is well-defined in those environments in which it is well-typed and is value-defined.

A result of this definition is that the type of the meaning of an expression in an environment is always the same as the type part of the expression when evaluated in the corresponding type-environment:

$$\vdash [\![E]\!]^{\mathcal{M}} \,;\, \imath \subseteq \Upsilon \,;\, [\![E]\!]^{\mathcal{T}} \,.$$

5.2 Identifier

An identifier is a name used to refer to a variable. Variables in Z are mathematical variables and are not the same as the programming variables used in programming languages. Z variables denote values which depend on their environment.

Abstract Syntax

IDENT = VARNAME

Note: A variable name is composed of a *base-name* suffixed by any number of *decorations*.

Representation and transformation

Production	Concrete	Abstract
VarName	n	n

Type The type of an identifier is the type to which the identifier is mapped in the type-environment:

$$[\![n]\!]^{\mathcal{T}} = (\_\ n).$$

Note: An identifier is well-typed only if it is in the domain of the type environment.

Value The value of an identifier is the element mapped to the identifier in the environment:

$$[\![n]\!]^{\mathcal{V}} = (\_\ n) \mathbin{;} v.$$

Note: An identifier is value-defined only if it is in the domain of the type environment.

5.3 Generic Instantiation

The generic instantiation $n\ [s_1, \ldots, s_n]$ is the instantiation of the generically declared variable n by by the list of set expressions $s_1, \ldots, s_n$. Each element of the instantiation list gives a value to a generic parameter of the generic definition.

If the list of generic parameters is omitted in the representation form, they are inferred from the typing information in the context of use. The implicit parameters are the maximal sets of the appropriate type, which must be uniquely determined by the typing rules.

Abstract Syntax A generic instantiation is constructed from a variable name and a list of expressions.

GENINST = VARNAME [EXP, EXP, ..., EXP]

Represeutation and transformation There are three ways of instantiating generically declared variables: by a parameter list, by infix or by prefix means.

Production	Concrete	Abstract
VarName,'[',Expression,{',',Expression}']'	$n_{[s_1,s_2,\ldots,s_n]}$	$n\ [[\![s_1]\!]^{\mathcal{E}}, [\![s_2]\!]^{\mathcal{E}}, \ldots, [\![s_n]\!]^{\mathcal{E}}]$
Expression1, InGen,Expression	$x_1 \psi x_2$	$(\_\psi\_)\ [[\![x_1]\!]^{\mathcal{E}}, [\![x_2]\!]^{\mathcal{E}}]$
PreGen,Expression5	ϕx	$(\phi\_)\ [[\![x]\!]^{\mathcal{E}}]$

Note: The expression $x_1 \psi x_2$, where ψ is an infix generic symbol is the variable declared as $(\_\psi\_)$ when instantiated with the parameter list $[x_1, x_2]$. When ψ is a prefix generic symbol then ϕx is the variable declared as $(\phi\_)$ when instantiated with the parameter list $[x]$.

Type The type of a generic instantiation $n\ [s_1, \ldots, s_n]$ is obtained by applying the function corresponding to the generic type of the variable name n in the environment to the types of the actual parameters $s_1, \ldots, s_n$:

$$[\![n[s_1, \ldots, s_n]\ ]\!]^{\mathcal{T}} \quad = \quad (\_\ n) \bullet \langle [\![s_1\ ]\!]^{\mathcal{T}}, \ldots, [\![s\ ]\!]^{\mathcal{T}} \rangle$$

Note: A generic instantiation is well-typed only if the variable name is in the domain of the type environment and if there is a correct number of set-typed parameters.

Value The value of a generic instantiation $n\ [s_1, \ldots, s_n]$ is obtained by applying the function corresponding to the generic meaning of the variable name n in the environment to the meanings of the actual parameters $s_1, \ldots, s_n$:

$$[\![n[s_1, \ldots, s_n]\ ]\!]^{\mathcal{V}} \quad = \quad ((\_\ n) \bullet \langle [\![s_1\ ]\!]^{\mathcal{M}}, \ldots, [\![s_n\ ]\!]^{\mathcal{M}} \rangle)\ ;\ v$$

Note: A generic instantiation is value-defined only if it is well-typed and all its parameters are value defined.

5.4 Number Literal

A number literal is an entity whose representation denotes its value in the world of integers.

Abstract Syntax

NUMBERL = NUMBER

Note: A number is a sequence of digits

Representation and transformation

Production	Concrete	Abstract
Number	i	$\mathbf{i}$

Type The type of a number literal is the given type of the integers.

$$[\![\mathbf{i}\,]\!]^{\mathcal{T}} \;=\; \mathbb{Z}^{\circ} \,;\, givenT$$

Note: A number literal is always well-typed

Value The value of a number literal is its representation.

$$[\![\mathbf{i}\,]\!]^{\mathcal{T}} \;=\; \mathbf{i}^{\circ}$$

Note: A number literal is always valne-defined

5.5 String Literal

A string literal is an entity whose representation denotes its value in the world **S** of strings of characters.

Abstract Syntax

```
STRINGL = STRING
```

Note: A string is a sequence of characters.

Representation and transformation

Production	Concrete	Abstract
String	a	$\mathbf{a}$

Type The type of a string literal is the set **S** of strings.

$$[\![\mathbf{a}\ ]\!]^{\mathcal{T}} \;=\; \mathbf{S}^{\circ}\ ;\ given\,T$$

Note: A string literal is always well-typed.

Value The value of a string literal is its representation.

$$[\![\mathbf{a}\ ]\!]^{\mathcal{T}} \;=\; \mathbf{a}^{\circ}$$

Note: A string literal is always value-defined.

5.6 Set Extension

A set extension $\{x_1, \ldots, x_n\}$ is a set containing exactly those elements denoted by $x_1, \ldots, x_n$. Since a set is characterised by its members, the order and multiplicity of elements in $x_1, \ldots, x_n$ is of no consequence.

Abstract Syntax A set extension is constructed from a list of expressions.

$$\mathsf{SETEXTN} = \{\mathsf{EXP}, \mathsf{EXP}, \ldots, \mathsf{EXP}\}$$

Representation and transformation There are three kinds of sets which can be constructed by extension: simple sets, sequences and bags.

Production	Concrete	Abstract
'{' ,Expression0,{',',Expression0} , '}'	$\{\ x_1, x_2, \ldots, x_n\ \}$	$\{\llbracket x_1 \rrbracket^{\mathcal{E}}, \llbracket x_2 \rrbracket^{\mathcal{E}}, \ldots, \llbracket x_n \rrbracket^{\mathcal{E}}\}$
'⟨' ,Expression0,{',',Expression0} , '⟩'	$\langle\ x_1, x_2, \ldots, x_n\ \rangle$	$\llbracket \{(1, x_1), (2, x_2), \ldots, (n, x_n)\} \rrbracket^{\mathcal{E}}$
'⟦' ,Expression0,{',',Expression0} , '⟧'	$\llbracket\ x_1, x_2, \ldots, x_n\ \rrbracket$	$\llbracket \{(x_1, 1)\} \uplus \{(x_2, 1)\} \uplus \ldots \uplus \{(x_n, 1)\} \rrbracket^{\mathcal{E}}$

Note: The expression $\langle\ x_1, x_2, \ldots, x_n\ \rangle$ defines an explicit construction of a sequence, which can be regarded as an ordered collection of its constituents. A sequence is modelled as a partial function mapping the Natural numbers $1, \ldots, n$ to the expressions $x_1, x_2, \ldots, x_n$ respectively.

Note: The expression $\llbracket\ x_1, x_2, \ldots, x_n\ \rrbracket$ defines an explicit construction of a bag. A bag is a collection of possibly multiply-occurring elements. A bag is modelled as a partial function mapping constituent expressions to the number of times they occur within the bag.

Type The type of a set extension $\{x_1, \ldots, x_n\}$ is the power set type of the common type of $x_1, \ldots, x_n$.

$$\llbracket \{x_1, \ldots, x_n\} \rrbracket^{\mathcal{T}} = (\llbracket x_1 \rrbracket^{\mathcal{T}} \cap \ldots \cap \llbracket x_n \rrbracket^{\mathcal{T}})\ ;\ powerT$$

Note: A set extension $\{x_1, \ldots, x_n\}$ is well typed only if all of the expressions $x_1, x_2, \ldots, x_n$ have the same type.

Note: If t represents the common type of $x_1, x_2, \ldots, x_n$, then $\mathsf{P}\ t$ represents the type of the set $\{\ x_1, x_2, \ldots, x_n\ \}$. $\mathsf{P}(\mathsf{Z} \times t)$ represents the type of the sequence $\langle\ x_1, x_2, \ldots, x_n\ \rangle$ and $\mathsf{P}(t \times \mathsf{Z})$ represents the type of the bag $\llbracket\ x_1, x_2, \ldots, x_n\ \rrbracket$.

Value The value of a set extension $\{x_1, \ldots, x_n\}$ is the set of the values of $x_1, \ldots, x_n$:

$$[\![\{x_1, \ldots, x_n\}]\!]^\nu \quad = \quad \langle [\![x_1]\!]^\nu, \ldots, [\![x_n]\!]^\nu \rangle \mathbin{;} \{\ldots\}$$

Note: A set extension $\{ x_1, x_2, \ldots, x_n \}$ is value-defined only if all of $x_1, x_2, \ldots, x_n$ are value-defined.

Note: Two sets $\{ x_1, x_2, \ldots, x_n \}$ and $\{ y_1, y_2, \ldots, y_m \}$ are equal if and only if for all x_i there exists y_j such that $x_i = y_j$, $1 \leq i \leq n$ and for all y_j there exists x_k such that $y_j = x_k$, $1 \leq j \leq m$

5.7 Set Comprehension

The set comprehension $\{ St \bullet x \}$ is a set which contains exactly those elemeuts denoted by the expression x when evaluated in each enrichment of the current environment by the schema text St.

Abstract Syntax A set comprehension is constructed from a schema text and an expression.

SETCOMP = {SCHEMATEXT • EXP}

Representation and transformation There are two types of set which can be constructed by comprehension: a simple set (for which the expression part is optional) and a lambda expression.

Production	Concrete	Abstract
'{' ,SchemaText, '•' ,Expression0, '}'	$\{St \bullet x\}$	$\{[St]^{ST} \bullet [x]^{\mathcal{E}}\}$
'{' ,SchemaText, '}'	$\{St\}$	$\{[St]^{ST} \bullet [(St)^{\times}]^{\mathcal{E}}\}$
'λ' ,SchemaText, '•' ,Expression	$\lambda\, St \bullet x$	$\{[St]^{ST} \bullet ([(St)^{\times}]^{\mathcal{E}}, [x]^{\mathcal{E}})\}$

Note: If the expression part of the set comprehension is omitted then the default is the characteristic tuple of the schema text.

Note: A lambda expression denotes a fuuction. The parameter is the characteristic tuple of the SchemaText. The domain is defined by the property of the SchemaText. The value of the function for a given parameter is defined by the value of the Expression with respect to the value of the parameter.

Type The type of a set comprehension $\{St \bullet x\}$ is the power set type of the type of x in the type-environment enriched by the declaration St:

$$[\![\{St \bullet x\}]\!]^{\mathcal{T}} = (\!| St |\!)^{\mathcal{T}} \mathbin{;} [\![x]\!]^{\mathcal{T}} \mathbin{;} powerT$$

Note: A set comprehension $\{St \bullet x\}$ is well-typed only if St is well-typed and x is well-typed in the current type-environment enriched by St.

Value The value of a set comprehension $\{St \bullet x\}$, is the set of the values denote by the expression x in each of the enrichments of the environment by the schema text St:

$$[\![\{St \bullet x\}]\!]^{\mathcal{V}} = {}^{\wedge}((\!| St |\!)^{\mathcal{M}} \mathbin{;} [\![x]\!]^{\mathcal{V}})$$

Note: A set comprehension is always value-defined.

5.8 Power Set

The power set **P** ***s*** is the set of all subsets of the set ***s***.

Abstract Syntax A power set is constructed from an expression.

```
POWERSET = P EXP
```

Representation and transformation

Production	Concrete	Abstract
'P' ,Expression5	**P** *s*	$\mathsf{P}[s]^{\mathcal{E}}$

Type The type of the power set **P** ***s*** is the power set type of the type of the set ***s***.

$$[\![\mathbf{P}\ \boldsymbol{s}]\!]^{\mathcal{T}} = ([\![\boldsymbol{s}]\!]^{\mathcal{T}} \rhd Ptype)\ ;\ powerT$$

Note: A power set **P** ***s*** is well typed only if ***s*** has power set type.

Note: If **P** *t* represents the type of the set *s*, then **P P** *t* represents the type of **P** *s* - it is a set of sets. So, the type of the elements of **P** *s* is the type of *s*.

Value The value of the power set **P** ***s*** is the set of all the subsets of the value of ***s***:

$$[\![\mathbf{P}\ \boldsymbol{s}]\!]^{\mathcal{V}} = [\![\boldsymbol{s}]\!]^{\mathcal{V}}\ ;\ \mathbf{P}$$

Note: A power set **P** ***s*** is value-defined only if the expression ***s*** is value-defined.

5.9 Tuple

A tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is an ordered collection of the elements $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. The elements $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are not required to have the same type.

> **Note:** Note that the tuples (a, b, c) and $((a, b), c)$ are distinct: the first contains three elements a, b, c whereas the second contains two elements $(a, b), c$. The expression (a) is not a tuple; it is the expression a within parentheses.

Abstract Syntax A tuple is constructed from a list of two or more expressions.

$$\mathsf{TUPLE} = (\mathsf{EXP}, \mathsf{EXP}, \ldots, \mathsf{EXP}, \mathsf{EXP})$$

Representation and transformation

Production	Concrete	Abstract
'(' ,Expression0, ',' ,Expression0,{',',Expression0} , ')'	$(x_1, \ldots, x_n)$	$(\llbracket x_1 \rrbracket^{\mathcal{E}}, \ldots, \llbracket x_n \rrbracket^{\mathcal{E}})$

Type The type of a tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is the Cartesian product type formed from the types of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$:

$$\llbracket (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \rrbracket^{\mathcal{T}} = \langle \llbracket \boldsymbol{x}_1 \rrbracket^{\mathcal{T}}, \ldots, \llbracket \boldsymbol{x}_n \rrbracket^{\mathcal{T}} \rangle \; ; \; cproductT$$

> **Note:** A tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is well-typed only if all of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are well-typed.

Value The value of a tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is the tuple formed from the values of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$:

$$\llbracket (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \rrbracket^{\mathcal{V}} = \langle \llbracket \boldsymbol{x}_1 \rrbracket^{\mathcal{V}}, \ldots, \llbracket \boldsymbol{x}_n \rrbracket^{\mathcal{V}} \rangle$$

> **Note:** A tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is value-defined only if all of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are value-defined.

> **Note:**
>
> Two tuples ($x_1, x_2, \ldots, x_n$) and ($y_1, y_2, \ldots, y_m$) are equal if and only if $x_i = y_i$, $1 \leq i \leq n = m$

> **Note:** If $x_i \in s_i$ for $1 \leq i \leq n$, then the tuple ($x_1, x_2, \ldots, x_n$) is an element of $s_1 \times s_2 \times \ldots \times s_n$.

5.10 Cartesian Product

The expression $s_1 \times \ldots \times s_n$ is the Cartesian product of the sets $s_1, \ldots, s_n$.

> **Note:** Cartesian products with different numbers of terms are distiuct.

Abstract Syntax A Cartesian Product is constructed from two or more expressions.

$$\mathsf{PRODUCT} = \mathsf{EXP} \times \mathsf{EXP} \times \ldots \times \mathsf{EXP} \times \mathsf{EXP}$$

Representation and transformation

Production	Concrete	Abstract
Expression2, '×' ,Expression2,{'×',Expression2}	$s_1 \times s_2 \times \ldots \times s_n$	$[s_1]^{\mathcal{E}} \times \ldots \times [s_n]^{\mathcal{E}}$

Type The type of a Cartesian product $s_1 \times \ldots \times s_n$ is the power set type of the Cartesian product type of the list of the underlying types of the elements $s_1, \ldots, s_n$.

$$[\![s_1 \times \ldots \times s_n]\!]^{\mathcal{T}} = \langle [\![s_1]\!]^{\mathcal{T}} ; powerT^{-1}, \ldots, [\![s_n]\!]^{\mathcal{T}} ; powerT^{-1} \rangle ; cproductT ; powerT$$

> **Note:** A Cartesian product $s_1 \times \ldots \times s_n$ is well-typed only if all of the elements $(s_1, \ldots, s_n)$ have power set types.

Value The value of a Cartesian product $s_1 \times \ldots \times s_n$ is the Cartesian product of the values of the sets $(s_1, \ldots, s_n)$:

$$[\![s_1 \times \ldots \times s_n]\!]^{\mathcal{V}} = \langle [\![s_1]\!]^{\mathcal{V}}, \ldots, [\![s_n]\!]^{\mathcal{V}} \rangle ; \times$$

> **Note:** A Cartesian product $s_1 \times \ldots \times s_n$ is value-defined exactly only if all of the sets $s_1, \ldots, s_n$ are value-defined.

5.11 Tuple Selection

The tuple selection $t.i$ is the ith element in the tuple t.

Abstract Syntax A tuple selection is constructed from an expression and a number literal.

TUPLESELECTION = EXP . NUMBERL

Representation and transformation

Production	Concrete	Abstract
Expression5, '.' ,Numberl	$t.i$	$[\![t]\!]^{\mathcal{E}}.i$

Type The type of a tuple selection $t.i$ is the type of the ith element of the tuple t.

$$[\![t.i]\!]^{\mathcal{T}} = [\![t]\!]^{\mathcal{T}} \mathbin{;} cproductT^{-1} \mathbin{;} \pi_i$$

Note: The tuple selection $t.i$ is well-typed only if t has a Cartesian product type with at least i elements.

Value The value of a tuple selection $t.i$ is the value of the ith element of the tuple t.

$$[\![t.i]\!]^{\mathcal{V}} = [\![t]\!]^{\mathcal{V}} \mathbin{;} \pi_i$$

Note: The tuple selection $t.i$ is value-defined only if t has the value of a tuple with at least i elements.

5.12 Binding Extension

A binding extension $\lfloor n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \rfloor$ is the binding which maps the names $n_1, \ldots, n_m$ to the values of the expressions $x_1, \ldots, x_m$ respectively.

Abstract Syntax A binding extension is constructed from a list of names and expressions.

$$\mathsf{BINDINGEXTN} \;=\; \lfloor \mathsf{VARNAME} \leadsto \mathsf{EXP}, \ldots, \mathsf{VARNAME} \leadsto \mathsf{EXP} \rfloor$$

Representation and transformation

Production	Concrete	Abstract
'⦇ ',VarName,'⤳',Expression0, {',',VarName,'⤳',Expression0},'⦈ '	$\lfloor n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \rfloor$	$\lfloor n_1 \leadsto [\![x_1]\!]^{\mathcal{E}}, \ldots, n_m \leadsto [\![x_m]\!]^{\mathcal{E}} \rfloor$

Type The type of a binding extension $\lfloor n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \rfloor$ is the schema type of the signature constructed from the mapping of the names $n_1, \ldots, n_m$ to the types of the expressions $x_1, \ldots, x_m$.

$$[\![\lfloor n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \rfloor\ ]\!]^{\mathcal{T}} \;=\; \langle \langle n_1{}^{\circ}, [\![x_1]\!]^{\mathcal{T}} \rangle, \ldots, \langle n_m{}^{\circ}, [\![x_m]\!]^{\mathcal{T}} \rangle \rangle \;;\; \{\ldots\} \;;\; \mathit{schemaT}$$

Note: A binding extension $\lfloor n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \rfloor$ is well-typed only if the expressions $x_1, \ldots, x_m$ are all well-typed, and if the mapping from names to types is functional.

Value The value of a binding extension $\lfloor n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \rfloor$ is the binding constructed from the mapping of the names $n_1, \ldots, n_m$ to the values of the expressions $x_1, \ldots, x_m$.

$$[\![\lfloor n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \rfloor\ ]\!]^{\mathcal{V}} \;=\; \langle \langle n_1^{\circ}, [\![x_1]\!]^{\mathcal{V}} \rangle, \ldots, \langle n_m^{\circ}, [\![x_m]\!]^{\mathcal{V}} \rangle \rangle \;;\; \{\ldots\}$$

Note: A binding extension $\lfloor n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \rfloor$ is value-defined only if the expressions $x_1, \ldots, x_m$ are all value-defined, and if the mapping from names to values is functional.

Note: Two bindings x and y with components $n_1, \ldots n_k$ are equal if and only if $x.n_i = y.n_i$, $1 \leq i \leq k$.

5.13 Theta Expression

The theta expression $\theta\ S$ is the binding whose type is constructed from tbe signature of S and whose value is the binding constructed from the mapping of the names of the signature to their values in the environment. The theta expression $\theta\ S\ ^q$ is the binding whose type is constructed from the signature of S and whose value is the binding constructed from the mapping of the names of the signature to the values in the environment of those names when decorated by ^{q}.

A θ-expression is a way of identifying a binding. A binding can be constrncted from variables in scope if for each named element in the binding, there is the same name in the environment denoting the same element.

Abstract Syntax A theta expression is constructed from a schema and an optional decoration.

```
THETAEXP = θ SCHEMA DECOR
         | θ SCHEMA
```

Note: The schema may itself be decorated. Thus the following are permitted: $\theta\ S\ ^q$ and $\theta\ (S^q)\ ^q$. Only non-generic schemas may be nsed in theta expressions

Representation and transformation

Production	Concrete	Abstract
'θ' ,BasicSch,Decoration	$\theta\ S\ ^q$	$\theta[S]^S\ ^q$
'θ' ,BasicSch	$\theta\ S$	$\theta[S]^S$

Type The type of $\theta\ S\ ^q$ is the schema type constrncted from the signatnre of S whose components, when decorated by ^{q}, have the same non-generic type in the environment:

$$\begin{aligned}
[\![\theta S\ ]\!]^T &= ((\!| S\ |\!)^{Ts} \cap \supseteq)\ ;\ schemaT \\
[\![\theta S^q\ ]\!]^T &= ((\!| S\ |\!)^{Ts} \cap \supseteq;\ ^\exists((\!| q\ |\!)^N \times 1))\ ;\ schemaT
\end{aligned}$$

Note: A theta expression is well-typed only when each of the decorated versions of the names of the signature of the schema are assigned non-generic types in the environment and they have the same type as those of the signature.

Note: The type of a theta expression $\theta\ S\ ^q$ is *not* the type taken from S decorated by ^{q}. The decoration ^{q} does *not* necessarily appear in the resulting type. The use of the schema is to identify the type of the resulting binding. Decoration is used only to identify which names to look up in the environment; thus $\theta\ S\ ^r$ and $\theta\ S\ ^q$ are of the same type even if ^{r} and ^{q} are different decorations.

Value The value of the theta expression $\boldsymbol{\theta}\ \boldsymbol{S}\ ^{q}$ is a bindiug of the names of the components of $\boldsymbol{S}$ to the values of the names, when decorated by ^{q}, in the environment:

$$
\begin{aligned}
[\![\boldsymbol{\theta} S\,]\!]^{\nu} &= \Upsilon\,;\,(\!|S\,|\!)^{Ts}\,;\,schemaT\,;\,Elm \cap \sqsupseteq;\ V \\
[\![\boldsymbol{\theta} S^{q}\,]\!]^{\nu} &= \Upsilon\,;\,(\!|S\,|\!)^{Ts}\,;\,schemaT\,;\,Elm \cap \sqsupseteq;\ ^{\exists}((\!|q\,|\!)^{\mathcal{N}} \times v)
\end{aligned}
$$

Note: A well-typed theta expression is always value-defined. The value of the theta-expression does not have to satisfy the property of the schema.

5.14 Schema Expression

A schema expression S is the set of bindings defined by the schema. These bindings have as their type the schema-type constructed from the signature of S and they satisfy its property.

Abstract Syntax A schema expression is constructed from a schema.

$$\mathsf{SCHEMAEXP} ::= \mathsf{SCHEMA}$$

Representation and transformation

Production	Concrete	Abstract
Schema	S	$[S]^{S}$

Type The type of a schema expression S is the power set type of the schema type constructed from the signature of the schema S:

$$[\![S]\!]^{T} = (\!|S|\!)^{T_S} \,;\, schemaT \,;\, powerT$$

Note: A schema expression S is well-typed only if the schema S is well-typed.

Note: The type of a schema expression is not in the range of $schemaT$: it is in the range of $schemaT \,;\, powerT$. The relationship between $(\!|\ |\!)^{T_S}$ and $[\![\ ]\!]^{T}$ is that of $schemaT \,;\, powerT$.

Value The value of a schema expression S is the set of bindings defined by the schema S:

$$[\![S]\!]^{V} = {}^{\Lambda}((\!|S|\!)^{M_S} \,;\, V)$$

Note: A schema expression S is always value-defined.

5.15 Binding Selection

The binding selection $b.n$ is the element to which the name n is mapped in the binding b.

Abstract Syntax A binding selection is constructed from a binding and a name.

BINDSELECTION = EXP . VARNAME

Representation and transformation

Production	Concrete	Abstract
Expression5, '.' ,VarName	$b.n$	$[b]^{\mathcal{E}}.n$

Type The type of a binding selection $b.n$ is the type to which the name n is mapped in the signature used to construct the schema type of the binding b:

$$[\![b.\, n]\!]^{\mathcal{T}} \;=\; [\![b]\!]^{\mathcal{T}} \;;\; schemaT^{-1} \;;\; (\_\; n)$$

Note: A binding selection $b.n$ is well-typed only if the type of b is a schema type; and the name n is in the domain of the signature from which the schema type is constructed.

Value The value of a binding selection $b.n$ is the value to which the name n is mapped in the binding b:

$$[\![b.\, n]\!]^{\mathcal{V}} \;=\; [\![b]\!]^{\mathcal{V}} \;;\; (\_\; n)$$

Note: A binding selection $b.n$ is value-defined only if the binding b is value-defined and the name n is in its domain.

5.16 Function Application

The function application $f\ x$ is the result of applying the function f to the argument x.

Abstract Syntax A function application is constructed from two expressions, a function and its argument.

FUNCTAPP = EXP(EXP)

Representation and transformation There are four ways of representing a function application: a normal form, an infix form, a superscript and a postfix form. For functions declared for use in postfix or infix form, underscores indicate the positions of the operands. The complete name of such a function includes the nndersrores and surrounding parentheses which are omitted when the operands are supplied in the form defined in the declaration.

Production	Concrete	Abstract
Expression4,Expression5	$f\ x$	$[\![f]\!]^{\mathcal{E}}([\![x]\!]^{\mathcal{E}})$
Expression2, InFun,Expression3	$x \phi y$	$(\_\phi\_)[\![(x,y)]\!]^{\mathcal{E}}$
Expression5,$^{\text{Expression0}}$	R^x	$(\mathit{iter}[\![x]\!]^{\mathcal{E}})([\![R]\!]^{\mathcal{E}})$
Expression5, PostFun,	$x\phi$	$(\_\phi)[\![x]\!]^{\mathcal{E}}$

Note: The function application $x\ \phi\ y$ is the infix application of the fuuction $(\_\phi\_)$ applied to the pair of arguments (x, y).

Note: The function application R^x denotes the x-iteration of the relation R; it is an abbreviation of the expression $\mathit{iter}\ x\ R$.

Note: The function application $x\phi$ is the postfix application of the functiou $(\_\phi)$ applied to the argument x.

Type In the expression $f(x)$ the type of f must be the power set type of the Cartesian product type of a 2-tuple of types, and the type of the argument x must be the first type iu this tuple; the type of $f(x)$ is the second type in the tuple.

$$[\![f(x)]\!]^{\mathcal{T}} = \left([\![f]\!]^{\mathcal{T}} ; \mathit{powerT}^{-1} ; \mathit{cproductT}^{-1} ; \{-\}\right) \bullet [\![x]\!]^{\mathcal{T}}$$

Note: The function application $f(x)$ is well-typed only if the type of f is a power set type of a pair of types with the first type in the pair the same as the type of x.

Note: If we evaluate the type of f, we get essentially a set of pairs, where each pair comprises the type of an argument and the type of its result. If we next evaluate the type of the particular argument x, then we can simply use the type of f as a function to look up the type of the result corresponding to x. We say the the type of f is essentially a set of pairs, because we must 'undo' the type constructors.

Value The value of a function application $f(x)$ is given by applying the value of f to the value of the argument x:

$$[\![f(x)]\!]^{\mathcal{V}} \supseteq {}^{\wedge}([\![f]\!]^{\mathcal{V}} \bullet [\![x]\!]^{\mathcal{V}}) \,;\, (-)^{-1}$$

Note: A well-typed function application $f(x)$ is defined if both f and x are defined and if there is a unique w such that $(x, w) \in f$.

Note: In Z, a function is modelled by its graph, which is a set of pairs; the first element of each pair representing an argument, and the second the result for that argument. For the function application $f(x)$ to be defined, f has only to be functional in the value of x. Providing that x evaluates in the environment ρ to a value v, and the value of f in ρ contains (v, w), and no other pair starting with v, then the expression $(f\ x)$ evaluates to w. So for a well-defined function application we would expect an equality of the following form:

$$[\![f(x)]\!]^{\mathcal{V}}{}_{\rho} = [\![f]\!]^{\mathcal{V}}{}_{\rho}\ ([\![x]\!]^{\mathcal{V}}{}_{\rho})$$

The promoted application of $f(x)$ provides a satisfactory meaning when the function application is well defined. It is necessary to decide what to do with $(f\ x)$ when f is not functional at x. This arises if there are several different pairs in the value of f, each having the same first element equal to the value of x or if there is none. The definition provided does not prescribe a value for a function applied outside its domain or where it is non-functional.

5.17 Definite Description

The definite description $\mu\, St \bullet x$ is the element denoted by x in the unique enrichment of the environment by the schema text St.

Abstract Syntax A definite description is constructed from a schema text and an expression.

$$\mathsf{DEFNDESCR} \;=\; \mu\,\mathsf{SCHEMATEXT} \bullet \mathsf{EXP}$$

Representation and transformation In the representation form for definite description, the expression part is optional.

Production	Concrete	Abstract
'μ' ,SchemaText, '$\bullet$' ,Expression	$\mu\, St \bullet x$	$\mu[St]^{ST} \bullet [x]^{\mathcal{E}}$
'μ' ,SchemaText	$\mu\, St$	$\mu[St]^{ST} \bullet [(St)^{\times}]^{\mathcal{E}}$

Note: If the expression part of the definite description is omitted then the default is the characteristic tnple of the schema text.

Type The type of the term $\mu\, St \bullet x$ is the type of x in the environment enriched by St:

$$[\![\mu\, St \bullet x\,]\!]^{\mathcal{T}} \;=\; (\!|St\,|\!)^{\mathcal{T}} \,;\, [\![x\,]\!]^{\mathcal{T}}$$

Note: The expression $\mu\, St \bullet x$ is well-typed only if St is well-typed and x is well-typed in the environment enriched by St.

Value The value of a definite description $\mu\, St \bullet x$ is the value of x in the unique enrichment of the environment by St:

$$[\![\mu\, St \bullet x\,]\!]^{\mathcal{V}} \;\supseteq\; {}^{\Lambda}((\!|St\,|\!)^{\mathcal{M}}) \,;\, \{-\}^{-1} \,;\, [\![x\,]\!]^{\mathcal{V}}$$

Note: A well-typed definite description $\mu\, St \bullet x$ is value-defined if there is exactly one defined enrichment of the environment by the schema text St and the expression x is value-defined in that enriched environment.

Note: This definition is not specific about the value of a badly formed definite description. If there is not an uniqne enrichment of the environment then the value is not prescribed by this standard.

5.18 Conditional Expression

The conditional expression **if** P **then** E_1 **else** E_2 **fi** evaluates to the expression E_1 if the predicate P is true, otherwise it evaluates to the expression E_2.

Abstract Syntax A conditional expression is constructed from a predicate and two expressions.

IFTHENELSE = **if** PRED **then** EXP **else** EXP **fi**

Representation and transformation

Production	Concrete	Abstract
'*If*'.Predicate,'*Then*',Expression ,'*Else*',Expression,'*Fi*'	*If P Then x Else y Fi*	$\mathbf{if}[P]^{\mathcal{P}}\mathbf{then}[x]^{\mathcal{E}}\mathbf{else}[y]^{\mathcal{E}}$

Type The type of the conditional expression **if** P **then** E_1 **else** E_2 **fi** is the common type of the expressions E_1 and E_2 wheu the predicate P is well-typed:

$$[\![\mathbf{if}\ P\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{fi}\ ]\!]^{\mathcal{T}} = \{\![P\ ]\!\}^{\mathcal{T}} \lhd ([\![x\ ]\!]^{\mathcal{T}} \cap [\![y\ ]\!]^{\mathcal{T}})$$

Note: The expression **if** P **then** E_1 **else** E_2 **fi** is well-typed only when the predicate P is well-typed and the expressions E_1 and E_2 both have the same type.

Value The value of the conditional expression **if** P **then** E_1 **else** E_2 **fi** is the value of the expressions E_1 when the predicate P is true, otherwise it is the value of the expression E_2:

$$[\![\mathbf{if}\ P\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{fi}\ ]\!]^{\mathcal{V}} = (\{\![P\ ]\!\}^{\mathcal{M}} \lhd [\![x\ ]\!]^{\mathcal{V}}) \cup (\{\![\neg P\ ]\!\}^{\mathcal{M}} \lhd [\![y\ ]\!]^{\mathcal{V}})$$

Note: The expression **if** P **then** E_1 **else** E_2 **fi** is value-defined only when the predicate P is true and the expression E_1 is value-defined or when the predicate $\neg P$ is true and the expression E_2 is value-defined.

5.19 Substitution

The substituted expression $b \odot E$ evaluates to the expression E in the environment enriched by the binding b.

Abstract Syntax A snbstituted expression is constrncted from a substitution expression and an expression.

$$\mathsf{EXPSUBSTITUTION} = \mathsf{EXP} \odot \mathsf{EXP}$$

Representation and trausformation

Production	Concrete	Abstract
Expression,'⊙' ,Expression	$b \odot x$	$\lfloor b \rfloor^{\mathcal{E}} \odot \lfloor x \rfloor^{\mathcal{E}}$

Type The type of the substitution $b \odot E$ is the type of the expression E in the type-environment enriched by the binding b.

$$[\![b \odot x]\!]^{\mathcal{T}} = \langle 1, [\![b]\!]^{\mathcal{T}} \,;\, schemaT^{-1} \rangle \,;\, \oplus \,;\, [\![x]\!]^{\mathcal{T}}$$

Note: The substitution $b \odot E$ is well-typed only if b has schema-type and the expression E is well-typed in the type-environment enriched by the binding b.

Value The value of the substitution $b \odot E$ is the value of the expression E in the environment enriched by the biuding b.

$$[\![b \odot x]\!]^{\mathcal{V}} = \langle 1, [\![b]\!]^{\mathcal{M}} \,;\, \langle \_, \_ \rangle \rangle \,;\, \ominus \,;\, [\![x]\!]^{\mathcal{V}}$$

Note: The substitution $b \odot E$ is value-defined only if b is value-defined and the expression E is value-defiued in the environment enriched by the binding b.

5.20 Free variables

Table 18: Expressions and their free variables

Expression	Free Variables
n	$\{n\}$
$n[s_1, \ldots, s_m]$	$\{n\} \cup (\phi_e s_1) \cup \ldots \cup (\phi_e s_m)$
i	$\{\ \}$
a	$\{\ \}$
$\{x_1, \ldots, x_m\}$	$(\phi_e x_1) \cup \ldots \cup (\phi_e x_m)$
$\{\ St \bullet x\ \}$	$\phi_d St \cup (\phi_e x \setminus \alpha St)$
$\mathbb{P}\ x$	$\phi_e x$
$(x_1, \ldots, x_m)$	$(\phi_e x_1) \cup \ldots \cup (\phi_e x_m)$
$s_1 \times \ldots \times s_m$	$(\phi_e s_1) \cup \ldots \cup (\phi_e s_m)$
$\langle\!\mid n_1 \leadsto x_1, \ldots, n_m \leadsto x_m \mid\!\rangle$	$(\phi_e x_1) \cup \ldots \cup (\phi_e x_m)$
θS^q	$(\phi_s S) \cup (\alpha S^q)$
$b.n$	$\phi_e b$
$t.i$	$\phi_e t$
$f\ x$	$(\phi_e f) \cup (\phi_e x)$
$\mu\ St \bullet x$	$\phi_d St \cup (\phi_e x \setminus \alpha St)$
S	$\phi_s S$
if P **then** x **else** y **fi**	$\phi_p P \cup \phi_e y \cup \phi_e y$
$b \odot x$	$\phi_e b \cup (\phi_e x \setminus \alpha b)$

5.21 Substitution

Table 19: Substitution into Expressions

Substitution	Equivalence
$b \odot n$	n
$b \odot \mathrm{n}[s_1, \ldots, s_m]$	$\mathrm{n}[b \odot s_1, \ldots, b \odot s_m]$
$b \odot \mathrm{i}$	i
$b \odot a$	z
$b \odot \{x_1, \ldots, x_n\}$	$\{b \odot x_1, \ldots, b \odot x_n\}$
$b \odot \{\ St \bullet u\ \}$	$\{(b \odot St) \bullet (b \setminus [St]) \odot \mathrm{u}\}$
$b \odot (\mathbb{P}\ u)$	$\mathbb{P}\ b \odot u$
$b \odot (x_1, \ldots, x_n)$	$(b \odot x_1, \ldots . b \odot x_n)$
$b \odot (s_1 \times \ldots \times s_n)$	$(b \odot s_1) \times \ldots \times (b \odot s_n)$
$b \odot (t.i)$	$(b \odot t).i$
$b \odot \lvert\ n_l \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m \rangle$	$\lvert\ n_1 \rightsquigarrow b \odot x_1, \ldots, n_m \rightsquigarrow b \odot x_m \rangle$
$b \odot \theta\ S\ '$	$\theta((b \odot S) \setminus b) \cup b \mid (b \odot S)$
$b \odot S$	$b \odot S$
$b \odot (c.n)$	$(b \odot c).n$
$b \odot (f\ x)$	$(b \odot f)\ (b \odot x)$
$b \odot (\mu\ St \bullet u)$	$(\mu(b \odot St) \bullet (b \setminus [St]) \odot u)$

6 Predicate

6.1 Introduction

A *Predicate* is the general form for expressing properties of the environment. These properties are relationships between the values of the variables in the environment. A predicate may be constructed in a number of ways. They may be sub-divided as follows:

Elements:

EQUALITY MEMBERSHIP

These denote the equality and membership relations between expressions.

Constants:

TRUTH FALSEHOOD

These denote the predicates true and false

Propositional Constructs:

NEGATION CONJUNCTION DISJUNCTION IMPLICATION EQUIVALENCE

These are predicates constructed using the propositional connectives.

Quantifications:

UNIVERSALQUANT EXISTSQUANT UNIQUEQUANT

These are predicates constructed using quantifiers.

Schema Predicate:

SCHEMAPRED

This is a predicate composed from a schema.

Substituted Predicate:

PREDSUBSTITUTION

This is a predicate evaluated following a substitution.

Abstract Syntax

```
PRED = EQUALITY
     | MEMBERSHIP
     | TRUTH
     | FALSEHOOD
     | NEGATION
     | DISJUNCTION
     | CONJUNCTION
     | IMPLICATION
     | EQUIVALENCE
     | UNIVERSALQUANT
     | EXISTSQUANT
     | UNIQUEQUANT
     | SCHEMAPRED
     | PREDSUBSTITUTION
```

The description of the meaning of a predicate can be split into two parts. The first gives rules for determining whether it is well typed or not. The second determines whether the predicate is supported in the environment. A predicate is *supported* in an environment if the values of the sub-expressions in the predicate are such that the predicate is true in that environment without necessarily considering whether it is well typed.

The combination of these two descriptions provides a meaning for predicates.

6.1.1 Type

Since in the abstract syntax of Z we already know that a certain construct is a predicate, when considering the type of a predicate the only matter of concern is whether it is well-typed. For this reason we represent the type function of a predicate as the set of type-environments in which it is well-typed.

$[\![\mathsf{PRED}]\!]^{\mathcal{T}} : \mathbf{P}\ Tenv$

Note: In contrast to predicates, when considering the type of an expression, there are two matters of concern: whether the expression well typed and if so what is its type. Hence the use of a partial function whose domain is the set of environments in which it is well-typed.

Note: The predicate $(x = y)$ is meaningless if the expressions x and y are uot of the same type. There is no meaningful way of comparing them. A predicate which is badly typed in all environments has a type function which evaluates to the empty set.

6.1.2 Value

The value function for a predicate is the set of environments in which it is supported:

$[\![\mathsf{PRED}]\!]^{\mathcal{V}} : \mathbf{P}\ Env$

Note: The predicate $\neg(x \in x)$ is supported in all environments. This is so because the axiom of regularity ensures that $x \in x$ is false and hence $\neg(x \in x)$ is true. On the other hand $x \in x$ is not well-typed so therefore $\neg(x \in x)$ is not well-typed.

6.1.3 Meaning

The environments in which a predicate holds (has a true meaning) are exactly those environments in which the predicate is supported and is well-typed.

$$\llbracket \mathsf{PRED} \rrbracket^{\mathcal{M}} == {}^{\exists}(\Upsilon^{-1})\llbracket \mathsf{PRED} \rrbracket^{\mathcal{T}} \cap \llbracket \mathsf{PRED} \rrbracket^{\mathcal{V}}$$

Note: As indicated in the note above the predicate $\neg(x \in x)$ is supported but not well-typed, hence it is false in all environments. The meaning of the predicate is the empty set: $\llbracket x \in x \rrbracket^{\mathcal{M}} = \{\,\}$.

6.2 Equality

Two expressions are equal if they have the same value and type.

Abstract Syntax An equality is constructed from two predicates.

EQUALITY = EXP = EXP

Representation and transformation

Production	Concrete	Abstract
Expression, '=' ,Expression	$[x = y]^{\mathcal{P}}$	$[x]^{\mathcal{E}} = [y]^{\mathcal{E}}$

Type An equality $x = y$ is well-typed in those environments in which the expressions x and y have the same type:

$$\{x = y\}^{\mathcal{T}} = \mathrm{dom}([\![x]\!]^{\mathcal{T}} \cap [\![y]\!]^{\mathcal{T}}).$$

Value An equality $x = y$ is supported in those environments in which the expressions x and y have the same values:

$$\{x_1 = x_2\}^{\mathcal{V}} = \mathrm{dom}([\![x_1]\!]^{\mathcal{V}} \cap [\![x_2]\!]^{\mathcal{V}}).$$

6.3 Membership

The membership relation $x \in y$ is true when the expression x is a member of the set denoted by the expression y.

Abstract Syntax A membership predicate is constructed from two expressions.

$$\mathsf{MEMBERSHIP} = \mathsf{EXP} \ \in \ \mathsf{EXP}$$

Representation and transformation There are three ways in which the membership predicate can be written: using the membership sign, using an infix relation and by using a prefix relation.

Production	Concrete	Abstract
Expression, '∈' ,Expression	$[\![x \in y]\!]^{\mathcal{P}}$	$[\![x]\!]^{\mathcal{E}} \in [\![y]\!]^{\mathcal{E}}$
PreRel,Expression	$x \rho y$	$[\![(x, y)]\!]^{\mathcal{E}} \in (\_ \rho \_)$
Expression, InRel,Expression	$\rho\ x$	$[\![x]\!]^{\mathcal{E}} \in (\rho \_)$

Note: The infix relation predicate $x\rho y$ is true if the expression x is related to the expression y by the relation ρ, i.e. if the tuple (x, y) is a member of the relation ρ.

Note: The prefix relation predicate ρx is trne if ρ holds for x, i.e. if x is a member of the set ρ.

Type A membership relation $x \in y$ is well-typed if and only if the type of the expression y is the power set type of that of the expression x:

$$[\![x \in y]\!]^{\mathcal{T}} = \operatorname{dom}([\![x]\!]^{\mathcal{T}} \mathbin{;} \mathit{powerT} \cap [\![y]\!]^{\mathcal{T}}).$$

Value A membership relation $x \in y$ is supported in all those environments in which the values of the expressions x is a member of the value of the expression y:

$$[\![x_1 \in x_2]\!]^{\mathcal{V}} = \operatorname{dom}([\![x_1]\!]^{\mathcal{V}} \cap [\![x_2]\!]^{\mathcal{V}} \mathbin{;} \ni).$$

6.4 Truth Literal

The truth literal **true** represents the predicate that always holds.

Abstract Syntax

TRUTH = **true**

Representation and transformation

Production	Concrete	Abstract
'*true*'	*true*	**true**

Type The truth literal **true** is well-typed in all environments:

$$\{\!\!\{ \mathbf{true} \}\!\!\}^{\mathcal{T}} = Tenv.$$

Value The truth literal **true** is supported in all environments:

$$\{\!\!\{ \mathbf{true} \}\!\!\}^{\mathcal{V}} = Env.$$

6.5 False Literal

The false literal **false** represents the predicate that never holds.

Abstract Syntax

$$\mathsf{FALSEHOOD} = \mathbf{false}$$

Representation and transformation

Production	Concrete	Abstract
'*false*'	*false*	**false**

Type The false literal **false** is well-typed in all environments:

$$\{\!\![false]\!\!\}^{\mathcal{T}} = Tenv.$$

Value The false literal **false** is supported in no environment:

$$\{\!\![false]\!\!\}^{\mathcal{V}} = \varnothing.$$

6.6 Negation

The negation $\neg P$ holds whenever the predicate P does not.

Abstract Syntax A negation is constructed from a predicate.

$$\mathsf{NEGATION} = \neg \mathsf{PRED}$$

Representation and transformation

Production	Concrete	Abstract
'¬' .BasicPred	$\neg P$	$\neg[P]^{\mathcal{P}}$

Type The negation $\neg P$ is well-typed exactly when the predicate P is well-typed:

$$[\![\neg\, P\,]\!]^{\mathcal{T}} = [\![P\,]\!]^{\mathcal{T}}.$$

Value The negation $\neg P$ is supported in those environments in which the predicate P is not supported:

$$[\![\neg\, P\,]\!]^{\mathcal{V}} = Env \setminus [\![P\,]\!]^{\mathcal{V}}.$$

6.7 Disjunction

The disjunction $P_1 \vee P_2$ holds whenever at least one of the predicates P_1 and P_2 holds.

Abstract Syntax A disjunction is constructed from two predicates.

DISJUNCTION = PRED ∨ PRED

Representation and transformation

Production	Concrete	Abstract
LogPred2, '∨' ,LogPred3	$P_1 \vee P_2$	$[P_1]^{\mathcal{P}} \vee [P_2]^{\mathcal{P}}$

Type The disjunction $P_1 \vee P_2$ is well-typed exactly when both predicates P_1 and P_2 are well-typed:

$$\{\!\!\{ P_1 \vee P_2 \}\!\!\}^{\mathcal{T}} = \{\!\!\{ P_1 \}\!\!\}^{\mathcal{T}} \cap \{\!\!\{ P_2 \}\!\!\}^{\mathcal{T}}.$$

Value The disjunction $P_1 \vee P_2$ is supported in those environments in which one or both of the predicates P_1, P_2 are supported:

$$\{\!\!\{ P_1 \vee P_2 \}\!\!\}^{\mathcal{V}} = \{\!\!\{ P_1 \}\!\!\}^{\mathcal{V}} \cup \{\!\!\{ P_2 \}\!\!\}^{\mathcal{V}}.$$

6.8 Conjunction

The conjunction $P_1 \land P_2$ holds if the predicates P_1 and P_2 both hold.

Abstract Syntax A conjunction is constructed from two predicates.

$$\mathsf{CONJUNCTION} = \mathsf{PRED} \land \mathsf{PRED}$$

Representation and transformation

Production	Concrete	Abstract
LogPred3, '∧' ,BasicPred	$P_1 \land P_2$	$[P_1]^P \land [P_2]^P$
InRelPred ,Rel,Expression,{Rel,Expression}	$x_1\ \rho_1\ x_2\ \rho_2\ \ldots \rho_{n-1}\ x_n$	$[x_1\ \rho_1\ x_2]^P \land [x_2\ \rho_2\ \ldots \rho_{n-1}\ x_n]^P$
Predicate,{Sep,Predicate}	$P_1 \mathsf{Sep}\, P_2 \mathsf{Sep} \ldots \mathsf{Sep}\, P_n$	$[P_1]^P \land [P_2]^P \land \ldots \land [P_n]^P$

Note: In predicates Sep is equivalent to ∧; such a conjunction has the lowest possible precedence and is equivalent to parenthesising the separate predicates and conjoining them.

Type The conjnnction of two predicates is well-typed exactly when both predicates are well-typed:

$$\{\!\![P_1 \land P_2]\!\!\}^{\mathcal{T}} = \{\!\![P_1]\!\!\}^{\mathcal{T}} \cap \{\!\![P_2]\!\!\}^{\mathcal{T}}.$$

Value The conjunction of two predicates is supported in those environments in which both predicates are supported:

$$\{\!\![P_1 \land P_2]\!\!\}^{\mathcal{V}} = \{\!\![P_1]\!\!\}^{\mathcal{V}} \cap \{\!\![P_2]\!\!\}^{\mathcal{V}}.$$

6.9 Implication

The implication $P_1 \Rightarrow P_2$ holds whenever the predicate P_1 does not hold or whenever the predicate P_2 does hold.

Abstract Syntax An implication is constructed from two predicates.

$$\mathsf{IMPLICATION} = \mathsf{PRED} \Rightarrow \mathsf{PRED}$$

Representation and transformation

Production	Concrete	Abstract
LogPred2, '⇒' ,LogPred1	$P_1 \Rightarrow P_2$	$[P_1]^{\mathcal{P}} \Rightarrow [P_2]^{\mathcal{P}}$

Type The implication $P_1 \Rightarrow P_2$ is well-typed exactly when both predicates P_1 and P_2 are well-typed:

$$\llbracket P_1 \Rightarrow P_2 \rrbracket^{\mathcal{T}} = \llbracket P_1 \rrbracket^{\mathcal{T}} \cap \llbracket P_2 \rrbracket^{\mathcal{T}}.$$

Value The implication $P_1 \Rightarrow P_2$ is true in those environments in which the negation of the predicate P_1 is supported or the predicate P_2 is supported:

$$\llbracket P_1 \Rightarrow P_2 \rrbracket^{\mathcal{V}} = \llbracket \neg P_1 \rrbracket^{\mathcal{V}} \cup \llbracket P_2 \rrbracket^{\mathcal{V}}.$$

6.10 Equivalence

An equivalence $P_1 \Leftrightarrow P_2$ holds whenever both predicates P_1 and P_2 hold or neither hold.

Abstract Syntax An equivalence is constructed from two predicates.

$$\mathsf{EQUIVALENCE} = \mathsf{PRED} \Leftrightarrow \mathsf{PRED}$$

Representation and transformation

Production	Concrete	Abstract
LogPred, '⇔' ,LogPred1	$P_1 \Leftrightarrow P_2$	$[P_1]^{\mathcal{P}} \Leftrightarrow [P_2]^{\mathcal{P}}$

Type The equivalence $P_1 \Leftrightarrow P_2$ is well-typed exactly when both predicates P_1 and P_2 are well-typed:

$$\llbracket P_1 \Leftrightarrow P_2 \rrbracket^{\mathcal{T}} = \llbracket P_1 \rrbracket^{\mathcal{T}} \cap \llbracket P_2 \rrbracket^{\mathcal{T}}.$$

Value The equivalence $P_1 \Leftrightarrow P_2$ is true in those environments in which both predicates P_1 and P_2 imply each other:

$$\llbracket P_1 \Leftrightarrow P_2 \rrbracket^{\mathcal{V}} = \llbracket P_1 \Rightarrow P_2 \rrbracket^{\mathcal{V}} \cap \llbracket P_2 \Rightarrow P_1 \rrbracket^{\mathcal{V}}.$$

6.11 Universal Quantification

The universally quantified predicate $\forall St \bullet P$ holds if the predicate P holds for all possible combinations of values of the components of the schema text St.

Abstract Syntax A universal quantification is constructed from a schema text and a predicate.

$$\mathsf{UNIVERSALQUANT} = \forall \mathsf{SCHEMATEXT} \bullet \mathsf{PRED}$$

Representation and transformation

Production	Concrete	Abstract
'∀' ,SchemaText, '•' ,Predicate	$\forall St \bullet P$	$\forall [St]^{\mathcal{ST}} \bullet [P]^{\mathcal{P}}$

Type A universal quantification $\forall St \bullet P$ is well-typed in those type-environments enriched by the schema text St in which the predicate P is well-typed:

$$\{\!\!\{ \forall St \bullet P \}\!\!\}^{\mathcal{T}} = \mathrm{dom}((\!\!| St |\!\!)^{\mathcal{T}} \rhd \{\!\!\{ P \}\!\!\}^{\mathcal{T}}).$$

Meaning A universal quantification $\forall St \bullet P$ is supported in those environments for which the predicate P is supported in every enrichment by the schema text St:

$$\{\!\!\{ \forall St \bullet P \}\!\!\}^{\mathcal{V}} = \{\!\!\{ \neg \exists St \bullet \neg P \}\!\!\}^{\mathcal{V}}.$$

Note: This semantic definition rests on the properties of de Morgan's Laws.

6.12 Existential Quantification

The existentially quantified predicate $\exists\, St \bullet P$ is true if the predicate P is true for at least one possible combination of values of the components of the schema text St.

Abstract Syntax An existential quantification is composed of a schema text and a predicate.

EXISTSQUANT = ∃SCHEMATEXT • PRED

Representation and transformation

Production	Concrete	Abstract
'∃' ,SchemaText, '•' ,Predicate	$\exists\, St \bullet P$	$\exists [St]^{ST} \bullet [P]^{P}$

Type An existential quantification $\exists\, St \bullet P$ is well-typed in those type-environments enriched by the schema text St in which the predicate P is well-typed:

$$\{\!\![\exists\, St \bullet P]\!\!\}^{\mathcal{T}} \;=\; \mathrm{dom}((St)^{\mathcal{T}} \rhd \{\!\![P]\!\!\}^{\mathcal{T}}).$$

Value An existential quantification $\exists\, St \bullet P$ is suported in those environments for which there exists an enrichment by the schema text St in which the predicate P is supported:

$$\{\!\![\exists\, St \bullet P]\!\!\}^{\mathcal{V}} \;=\; \mathrm{dom}((St)^{\mathcal{M}} \rhd \{\!\![P]\!\!\}^{\mathcal{V}}).$$

6.13 Unique Existential Quantification

The unique existentially quantified predicate $\exists_1 S \bullet P$ is true if the predicate P is true for exactly one possible combination of values of the components of tbe schema text S.

Abstract Syntax A unique existential quantification is constructed from a schema text and a predicate.

$$\mathsf{UNIQUEQUANT} \;=\; \exists_1 \, \mathsf{SCHEMATEXT} \bullet \mathsf{PRED}$$

Representation and transformation

Production	Concrete	Abstract
'$\exists_1$' ,SchemaText, '$\bullet$' ,Predicate	$\exists_1 St \bullet P$	$\exists_1 [St]^{ST} \bullet [P]^{P}$

Type A unique existential quantification $\exists_1 St \bullet P$ is well-typed in those type environmeuts that, when enriched by St, well-type P:

$$\llbracket \exists_1 St \bullet P \rrbracket^{\mathcal{T}} \;=\; \mathrm{dom}(\llbracket St \rrbracket^{\mathcal{T}} \rhd \llbracket P \rrbracket^{\mathcal{T}}).$$

Value A unique existential quantification $\exists_1 St \bullet P$ is supported in those environments for which there is exactly one enrichment by the schema text St which supports the predicate P.

$$\llbracket \exists_1 St \bullet P \rrbracket^{\mathcal{V}} \;=\; \mathrm{dom}(^{\Lambda}(\llbracket St \rrbracket^{\mathcal{M}} \rhd \llbracket P \rrbracket^{\mathcal{V}}) \,;\, \{-\}^{-1}).$$

6.14 Substitution

The substituted predicate $b \odot P$ is true whenever the predicate is true in the environment enriched by the binding b.

Abstract Syntax A substituted predicate is constructed from an expression and a predicate.

$$\mathsf{PREDSUBSTITUTION} = \mathsf{EXP} \odot \mathsf{PRED}$$

Representation and transformation

Production	Concrete	Abstract
Expression,'⊙' ,Predicate	$b \odot P$	$[b]^{\mathcal{E}} \odot [P]^{\mathcal{P}}$

Type The substituted predicate $b \odot P$ is well-typed in those type-environments in which the binding b is well-typed and when enriched by it the predicate P is well-typed:

$$\{\!\!\{ b \odot P \}\!\!\}^{\mathcal{T}} = \mathrm{dom}((1, [\![b]\!]^{\mathcal{T}} ; schemaT^{-1}) ; \oplus \rhd \{\!\!\{ P \}\!\!\}^{\mathcal{T}})$$

Value The substituted predicate $b \odot P$ is supported in those environments in which the binding b is value defined and when enriched by it support the predicate P:

$$\{\!\!\{ b \odot P \}\!\!\}^{\mathcal{V}} = \mathrm{dom}((1, [\![b]\!]^{\mathcal{M}} ; (\_,\_)) ; \oplus \rhd \{\!\!\{ P \}\!\!\}^{\mathcal{V}})$$

6.15 Free Variables

The free variables of predicates are detailed in the following table:

Table 20: Predicates and their free variables

Predicate	Free Variables
$\boldsymbol{x} = \boldsymbol{y}$	$(\phi_e \boldsymbol{x}) \cup (\phi_e \boldsymbol{y})$
$\boldsymbol{x} \in \boldsymbol{y}$	$(\phi_e \boldsymbol{x}) \cup (\phi_e \boldsymbol{y})$
true	$\{\,\}$
false	$\{\,\}$
$\neg \boldsymbol{P}$	$\phi_p \boldsymbol{P}$
$\boldsymbol{P} \lor \boldsymbol{Q}$	$(\phi_p \boldsymbol{P}) \cup (\phi_p \boldsymbol{Q})$
$\boldsymbol{P} \land \boldsymbol{Q}$	$(\phi_p \boldsymbol{P}) \cup (\phi_p \boldsymbol{Q})$
$\boldsymbol{P} \Rightarrow \boldsymbol{Q}$	$(\phi_p \boldsymbol{P}) \cup (\phi_p \boldsymbol{Q})$
$\boldsymbol{P} \Leftrightarrow \boldsymbol{Q}$	$(\phi_p \boldsymbol{P}) \cup (\phi_p \boldsymbol{Q})$
$\forall \boldsymbol{St} \bullet \boldsymbol{P}$	$\phi_d \boldsymbol{St} \cup (\phi_p \boldsymbol{P} \setminus \alpha \boldsymbol{St})$
$\exists \boldsymbol{St} \bullet \boldsymbol{P}$	$\phi_d \boldsymbol{St} \cup (\phi_p \boldsymbol{P} \setminus \alpha \boldsymbol{St})$
$\exists_1 \boldsymbol{St} \bullet \boldsymbol{P}$	$\phi_d \boldsymbol{St} \cup (\phi_p \boldsymbol{P} \setminus \alpha \boldsymbol{St})$
$\boldsymbol{S}$	$\phi_s \boldsymbol{S} \cup \alpha \boldsymbol{S}$
$\boldsymbol{b} \circledcirc \boldsymbol{P}$	$\phi_e \boldsymbol{b} \cup (\phi_p P \setminus \alpha \boldsymbol{b})$

Note: The free variables for the representation forms of these constructs are the same as for their abstract counterparts. For example: $\phi_e(x\ \rho\ y) = \phi_e((\boldsymbol{x}, \boldsymbol{y}) \in \rho = \phi_e(\boldsymbol{x}, \boldsymbol{y}) \cup \phi\rho$.

6.16 Substitution

Table 21: Substitution into Predicates

Substitution	Equivalence
$b \odot (u = v)$	$(b \odot u = b \odot v)$
$b \odot (u \in v)$	$(b \odot u \in b \odot v)$
$b \odot \mathbf{true}$	$\mathbf{true}$
$b \odot \mathbf{false}$	$\mathbf{false}$
$b \odot (\neg P)$	$\neg b \odot P$
$b \odot (P \lor Q)$	$b \odot P \lor b \odot Q$
$b \odot (P \land Q)$	$b \odot P \land b \odot Q$
$b \odot (P \Rightarrow Q)$	$b \odot P \Rightarrow b \odot Q$
$b \odot (P \Leftrightarrow Q)$	$b \odot P \Leftrightarrow b \odot Q$
$b \odot (\forall St \bullet Q)$	$\forall b \odot St \bullet (b \setminus [St]) \odot Q$
$b \odot (\exists St \bullet Q)$	$\exists b \odot St \bullet (b \setminus [St]) \odot Q$
$b \odot (\exists_1 St \bullet Q)$	$\exists_1 b \odot St \bullet (b \setminus [St]) \odot Q$
$b \odot S$	$b \odot S$

7 Declaration

7.1 Introduction

A declaration is the general form for introducing new variables into the environment. A declaration may be a SIMPLEDECL , which explicitly introduces new variables by name, or a SCHEMAINCL which introduces the components of a schema, or a COMPNDECL which can be any combination of the other two. A declaration may also be evaluated following a substitution.

Abstract Syntax

```
DECL = SIMPLEDECL
     | SCHEMAINCL
     | COMPNDECL
     | DECLSUBSTITUTION
```

When making declarations, the problem is not so much whether the declaration is well defined (although a declaration may fail to be defined). The problem is more to record the possible meanings of the newly declared name. A declaration denotes a *signature* and a set of situations.

7.1.1 Type

The type of a declaration is a signature which records the types of the elements denoted by the variables introduced:

$$(\!|\mathsf{DECL}|\!)^{\mathcal{T}} \;:\; \mathit{Tenv} \nrightarrow (\mathit{Name} \twoheadrightarrow \mathit{Type})$$

7.1.2 Meaning

A declaration introduces names to the environment which can assume certain values. These values are not fixed. We can consider the meaning of a declaration as a set of situations, each one recording one set of values for the new names. However, it is more convenient to consider the meaning of a declaration as a relation between environments and situations.

$$(\!|\mathsf{DECL}|\!)^{\mathcal{M}} \;:\; \mathit{Env} \leftrightarrow (\mathit{Name} \twoheadrightarrow \mathit{Elm})$$

The meaning of a declaration is partial because some declarations may fail — for example $n : s$ where s is undefined, or if s is an empty set.

We can prove the following:

$$\vdash (\!|D|\!)^{\mathcal{M}} \,;\, T \subseteq \Upsilon \,;\, (\!|D|\!)^{\mathcal{T}}$$

7.2 Simple Declarations

A simple declaration $n_1, \ldots n_m : s$ introduces variables named $n_1, \ldots n_m$ whose values are drawn from the set s.

Abstract Syntax A simple declaration is constructed from a list of names and an expression.

SIMPLEDECL = VARNAME, VARNAME, ..., VARNAME · EXP

Representation and transformation

Production	Concrete	Abstract
DeclName,{',',DeclName} , ':' ,Expression	$n_1, n_2, \ldots, n_m : s$	$n_1, n_2, \ldots, n_k : [s]^{\mathcal{E}}$

Type The type of the simple declaration $n_1, \ldots n_m : s$ is the signature constructed from the names $n_1, \ldots n_m$ and the underlying type of the set expression s.

$$(\!| n_1, \ldots, n_m : s |\!)^{\mathcal{T}} = [\![s]\!]^{\mathcal{T}} ; \langle (n_1{}^{\circ}, powerT^{-1}), \ldots, (n_m^{\circ}, powerT^{-1}) \rangle ; \{\ldots\}.$$

Note: The simple declaration $n_1, \ldots n_m : s$ is well-typed exactly when the expression s has power set type.

Meaning The meaning of the simple declaration $n_1, \ldots n_m : s$ is a relation from the environment to those situations which associate each of the names $n_1, \ldots n_m$ with one of the elements of the set expression s:

$$(\!| n_1, \ldots n_m : s |\!)^{\mathcal{M}} = [\![s]\!]^{\mathcal{M}} ; \langle (n_1{}^{\circ}, \ni), \ldots, (n_m{}^{\circ}, \ni) \rangle ; \{\ldots\}.$$

Note: The simple declaration $n_1, \ldots n_m : s$ is value-defined exactly when the expression s is a non-empty set.

Note: Suppose G is defined to be a given set. The type system defines the type of G to be $powerT(givenT\ \mathsf{N})$. In this way a declaration such as $x : G$ defines the type of x to be $givenT(G)$, as required:

7.3 Schema Inclusion

The schema inclusion S introduces the components of the schema and constrains their values as in the schema.

Abstract Syntax A schema inclusion is constructed from a schema.

$$\mathsf{SCHEMAINCL} = \mathsf{SCHEMA}$$

Representation and transformation

Production	Concrete	Abstract
Schema	S	$[S]^{S}$

Type The signature of a schema inclusion is the signature of the included schema:

$$(\!| S |\!)^{\mathcal{T}} = (\!| S |\!)^{\mathcal{T}_S}.$$

Note: The schema inclusion S is well-typed exactly when the schema S is well-typed.

Meaning The meaning of a schema inclusion is the relation from the environment to situations as defined in the meaning of the schema.

$$(\!| S |\!)^{\mathcal{M}} = (\!| S |\!)^{\mathcal{M}_S}.$$

Note: The schema inclusion S is value-defined exactly when the schema S is value-defined.

7.4 Compound Declarations

A compound declaration $D_1;\ D_2$ introduces the names in the declarations D_1 and D_2.

> **Note:** Variables may be introduced in local declarations more than once, provided that they have the same type. Repeated declarations do not add anything to the signature; however the constraint of the repeated declaration is conjoined with the constraints of all the other declarations.

Abstract Syntax A compound declaration is composed from a list of basic declarations.

COMPNDECL = DECL; DECL

Representation and transformation

Production	Concrete	Abstract
BasicDecl, ';' ,BasicDecl,{';',BasicDecl}	$D_1;\ D_2;\ \ldots;\ D_n$	$[D_1]^{\mathcal{D}};\ [D_2]^{\mathcal{D}};\ \ldots;\ [D_n]^{\mathcal{D}}$

Type The signature of a compound declaration $D_1;\ D_2$ is the join of the signatures of the declarations D_1 and D_2:

$$[\![D_1;\ D_2]\!]^{\mathcal{T}} \;=\; \langle [\![D_1]\!]^{\mathcal{T}}, [\![D_2]\!]^{\mathcal{T}} \rangle \mathbin{;} \sqcup.$$

> Note: This declaration is well-typed only if both of D_1 and D_2 are well-typed and their signatures are type compatible.

Meaning The value of a compound declaration is the set of bindings that, when restricted to the alphabet of each component, satisfy that component:

$$[\![D_1\ ;\ D_2]\!]^{\mathcal{M}} \;=\; \langle [\![D_1]\!]^{\mathcal{M}}, [\![D_2]\!]^{\mathcal{M}} \rangle \mathbin{;} \sqcup.$$

> **Note:** A componnd declaration $D_1;\ D_2$ is value-defined only if botb the declarations D_1 and D_2 are value-defined and if repeated declarations are value compatible.

> **Note:** Dnplicated declarations are significant in the evaluation of the characteristic tuple. The representative term can be a list of terms which form part of the top level tuple.

7.5 Substituted Declarations

The meaning of tbe substituted declaration $b \circledcirc D$ is the same as the meaning of the declaration D in the environment enriched by the binding b.

Abstract Syntax A substitnted declaration is composed of an expression and a declaration.

$$\mathsf{DECLSUBSTITUTION} = \mathsf{EXP} \circledcirc \mathsf{DECL}$$

Representation and transformation

Production	Concrete	Abstract
Expression,'⊚' ,Declaration	$b \circledcirc D$	$[\, b \,]^{\mathcal{E}} \circledcirc [\, D \,]^{\mathcal{D}}$

Type The signature of the substituted declaration $b \circledcirc D$ is the signature of the declaration D in the type-environment enriched by the binding b.

$$(\!| b \circledcirc D \,|\!)^{\mathcal{T}} = \langle 1, [\![b \,]\!]^{\mathcal{T}} \mathbin{;} schemaT^{-1} \rangle \mathbin{;} \oplus \mathbin{;} (\!| D \,|\!)^{\mathcal{T}}$$

A substituted declaration is well-typed only if the bindiug is well-typed and the declaration is well-typed in the enriched environment.

Meaning The situations of the substituted declaration $b \circledcirc D$ are the sitnations of the declaration D in the environment enriched by the binding b.

$$(\!| b \circledcirc D \,|\!)^{\mathcal{M}} = \langle 1, [\![b \,]\!]^{\mathcal{M}} \mathbin{;} \langle \_, \_ \rangle \rangle \mathbin{;} \oplus \mathbin{;} (\!| D \,|\!)^{\mathcal{M}}$$

7.6 Free Variables and Alphabet

The following tables define the free variables, alphabet and representative terms for declarations.

Table 22: Declarations and their free variables

Declaration	Free Variables	Alphabet
$n_1, \ldots, n_m : s$	ϕs	$\{n_1, \ldots, n_m\}$
S	$\phi_s S$	αS
$D_1;\ D_2$	$(\phi_d D_1) \cup (\phi_d D_2)$	$(\alpha D_1) \cup (\alpha D_2)$
$b \odot D$	$\phi_e b \cup (\phi_d D \setminus \alpha b)$	αD

Table 23: Declarations and their representative terms

Declaration	Representative Term
$n_1, \ldots, n_m : s$	$n_1, \ldots, n_m$
S	θS
$D_1;\ D_2$	D_1^λ, D_2^λ
$b \odot D$	D^λ

7.7 Substitution

The following table gives the semantic equivalence rules for substitution into declarations:

Table 24: Substitution into Declarations

Substitution	Equivalence
$b \odot n_1, \ldots, n_m : s$	$n_1, \ldots, n_m : b \odot s$
$b \odot (D_1;\ D_2)$	$b \odot D_1;\ b \odot D_2$

8 SchemaText

8.1 Introduction

A schema text is the general way of enriching the environment by the new names introdnced by a declaration and possibly constraining their values by a predicate. A SIMPLESCT consistsof a declaration and a CMPNDSCT consists of a declaration and a predicate.

Abstract Syntax

$$\begin{array}{lll} \mathsf{SCHEMATEXT} & = & \mathsf{SIMPLESCT} \\ & | & \mathsf{CMPNDSCT} \\ & | & \mathsf{SCTSUBSTITUTION} \end{array}$$

Given a certain environment, a schema text has the effect of defining a new environment in which the name is now known.

8.1.1 Type

The type of a schema text is a function from the old typ-environment to the new one in which the names of the constituent declaration are known:

$$\{\mathsf{SCHEMATEXT}\}^{\mathcal{T}} : Tenv \rightarrow Tenv$$

8.1.2 Meaning

The is represented as a relation between environments. for the same reason as the meaning of a declaration os represented by a relation.

$$\{\mathsf{SCHEMATEXT}\}^{\mathcal{M}} : Env \leftrightarrow Env$$

We can prove tbe following

$$\vdash \{St\}^{\mathcal{M}} ; \Upsilon \subseteq \Upsilon ; \{St\}^{\mathcal{T}}$$

8.2 Simple Schema Text

Abstract Syntax A simple schema text is constructed from a declaration.

SIMPLESCT = DECL

Representation and transformation

Production	Concrete	Abstract
Declaration	D	$[D]^{\mathcal{D}}$

Type A simple schema text D enriches the type-environment by the signature of the declaration D.

$$\{D\}^{\mathcal{T}} = \langle 1, (\!|D|\!)^{\mathcal{T}}\rangle \, ; \oplus.$$

Note: The simple schema text D is well-typed exactly when the declaration D is.

Meaning A simple schema text D enriches the environment by a situation of the declaration D.

$$\{D\}^{\mathcal{M}} = \langle 1, (\!|D|\!)^{\mathcal{M}}\rangle \, ; \oplus.$$

Note: The simple schema text D is well-defined exactly when the declaration D is.

8.3 Compound Schema Text

Abstract Syntax A compound schema text is constructed from a declaration and a predicate.

CMPNDSCT = DECL | PRED

Representation and transformation

Production	Concrete	Abstract
Declaration, '|' ,Predicate	$D \mid P$	$[D]^{\mathcal{P}} \mid [P]^{\mathcal{P}}$

Type A compound schema text $D \mid P$ enriches the type-environment by the signature of the declaration D.

$$\{D \mid P\}^{\mathcal{T}} = \{D\}^{\mathcal{T}} \rhd \{P\}^{\mathcal{T}}.$$

Note: The compound schema text $D \mid P$ is well-typed exactly when the declaration D is well-typed and the predicate P is well-typed in the environment enriched by the declaration D .

Meaning A compound schema text $D \mid P$ enriches the environment by a situation of the declaration D which makes the predicate P true.

$$\{D \mid P\}^{\mathcal{M}} = \{D\}^{\mathcal{M}} \rhd \{P\}^{\mathcal{M}}.$$

Note: The compound schema text $D \mid P$ is well-defined only when the declaration D is well-defined and the predicate P is true in at least one enrichment of the environment by the declaration D .

8.4 Substituted Schema Text

The meaning of the substituted schema text $b \odot St$ is the same as the meaning of the schema text St when evaluated in the environment enriched by the binding b.

Abstract Syntax A substituted schema text is constructed from an expression and a schema text.

$$\mathsf{SCTSUBSTITUTION} = \mathsf{EXP} \odot \mathsf{SCHEMATEXT}$$

Representation and transformation

Production	Concrete	Abstract
SctSubstitution	$b \odot St$	$[\![b]\!]^{\mathcal{E}} \odot [\![St]\!]^{\mathcal{ST}}$

Type A substituted schema text enriches the type-environment with the signature of the substituted schema constructed from the schema text.

$$\{\!| b \odot St \,|\!\}^{\mathcal{T}} = \{\!| b \odot (St) \,|\!\}^{\mathcal{T}}$$

Meaning A substituted schema text enriches the environment with the situations of the substituted schema constructed from the schema text.

$$\{\!| b \odot St \,|\!\}^{\mathcal{M}} = \{\!| b \odot (St) \,|\!\}^{\mathcal{M}}$$

8.5 Free Variables and Alphabet

Table 25: Schema Texts and their free variables

Schema Text	Free Variables	Alphabet
$\boldsymbol{D}$	$\phi_d \boldsymbol{D}$	αD
$\boldsymbol{D} \mid \boldsymbol{P}$	$\phi_d \boldsymbol{D} \cup (\phi_p \boldsymbol{P} \setminus \alpha \boldsymbol{D})$	αD
$b \odot \boldsymbol{St}$	$\phi_e b \cup (\phi_d \boldsymbol{St} \setminus \alpha b)$	αD

The characteristic tuple of a schema text is the tuple constructed from the representative terms of the declaration.

Table 26: Schema Texts and their characteristic tuples

Schema Text	Characteristic Tuple
D	(D^λ)
$D \mid P$	(D^λ)
$b \odot St$	(St^λ)

9 Schema

Abstract Syntax

```
SCHEMA = SDES
       | GENSDES
       | SCONSTRUCTION
       | SNEGATION
       | SDISJUNCTION
       | SCONJUNCTION
       | SIMPLICATION
       | SEQUIVALENCE
       | SPROJECTION
       | SHIDING
       | SUNIVQUANT
       | SEXISTSQUANT
       | SUNIQUEQUANT
       | SRENAMING
       | SCOMPOSITION
       | SDECORATION
       | SCHEMASUBSTITUTION
```

Z provides a number of schema operators that act on the underlying functions from names to type. In order to describe these operations, it is convenient to identify the type of a schema, not as an element of *TYPE*, but as a finite mapping from names to type. We shall call this the *signature* of a schema expression, and is written $(\!|\ |\!)^{\mathcal{T}s}$.

$$(\!|\mathsf{SCHEMA}|\!)^{\mathcal{T}s} : Tenv \twoheadrightarrow Signature$$

$$(\!|\mathsf{SCHEMA}|\!)^{\mathcal{M}s} : Env \leftrightarrow Situation$$

We can define the relation between the environment and the well-typed (thougb not necessarily well valued) bindings as follows:

$$(\!|S|\!)^{\mathcal{MT}s} == \Upsilon \mathbin{;} (\!|S|\!)^{\mathcal{T}s} \mathbin{;} T^{-1}$$

We can prove the following:

$$\vdash (\!|S|\!)^{\mathcal{M}s} \subseteq (\!|S|\!)^{\mathcal{MT}s}$$

9.1 Schema Designator

A schema designator is a schema name used to refer to schema. It may also contain a list of generic paramaters which instantiate a generically defined schema.

> **Note:** Since schema names have global scope there cannot be any overlap between the base names of variables and schema names in a specification.

Abstract Syntax A schema designator is constructed from a schema name.

SDES = WORD

Representation and transformation

Production	Concrete	Abstract
SchemaName	S	S

Type The signature of a schema reference is the signatnre of the type of the reference in the type-environment.

$$(\!|S|\!)^{Ts} = (1 \bullet S^{\circ}) \mathbin{;} powerT^{-1} \mathbin{;} schemaT^{-1}.$$

Note: A schema reference is well-typed only if it is in the domain of the type-environment.

Meaning The meaning of a schema reference is the relation constructed from the the meaning of the reference in the environment.

$$(\!|S|\!)^{\mathcal{M}s} = (1 \bullet S^{\circ}) \mathbin{;} \ni.$$

Note: A schema reference is well-defined only if it is in the domain of the environment.

9.2 Generic Schema Designator

A generic schema designator $S\ [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$ is reference to a generically defined schema S instantiated by the set paramaters $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$.

Abstract Syntax A generic schema designator is constructed from a schema name and a list of expressions.

$$\mathsf{GENSDES} = \mathsf{WORD}\ [\mathsf{EXP}, \ldots, \mathsf{EXP}]$$

Representation and transformation

Production	Concrete	Abstract
SchemaName,'[',Expression,{',',Expression}']'	$S_{[x_1,\ldots,x_n]}$	$S[[\![x_1]\!]^{\mathcal{E}}, \ldots, [\![x_n]\!]^{\mathcal{E}}]$

Type

$$(\![S[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]\,]\!)^{\mathcal{T}_S} = ((1 \bullet S^\circ) \bullet \langle x_1, \ldots, x_n \rangle) \mathbin{;} powerT^{-1} \mathbin{;} schemaT^{-1}.$$

Meaning

$$(\![S[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]\,]\!)^{\mathcal{M}_S} = ((1 \bullet S^\circ) \bullet \langle x_1, \ldots, x_n \rangle) \mathbin{;} \ni.$$

Note:

Generically defined schemas must be instantiated.

9.3 Schema Construction

A schema construction $\langle D \mid P \rangle$ is a schema whose signature is that of the declaration D and whose components satisfy the constraint of the declaration D and the predicate P.

Abstract Syntax A schema construction is composed from a declaration and a predicate.

$$\mathsf{SCONSTRUCTION} \ = \ \langle \mathsf{DECL} \mid \mathsf{PRED} \rangle$$

Representation and transformation

Production	Concrete	Abstract
'[' ,Declaration, '|' ,Predicate, ']'	$[D \mid P]$	$\langle [\![D]\!]^{\mathcal{D}} \mid [\![P]\!]^{\mathcal{P}} \rangle$
'[' ,Declaration, ']'	$[D]$	$\langle [\![D]\!]^{\mathcal{D}} \mid \mathsf{true} \rangle$

Type The signature of $\langle D \mid P \rangle$ is the same as that of the declaration D.

$$(\!| \langle D \mid P \rangle |\!)^{\mathcal{T}s} \ = \ (\!| D |\!)^{\mathcal{T}} \cap (\langle D \mid P \rangle^{\mathcal{T}} ; \supseteq).$$

Meaning The value of the schema expression constructed from $\langle D \mid P \rangle$ is a set of bindings. The bindings are constructed iu all enrichments of the environment by D which satisfy P:

$$(\!| \langle D \mid P \rangle |\!)^{\mathcal{M}s} \ = \ (\!| D |\!)^{\mathcal{M}} \cap (\langle D \mid P \rangle^{\mathcal{M}} ; \supseteq).$$

This is defined only in those environments in which the declaration D is defined and when enriched by it result in the predicate P heing well-typed.

9.4 Schema Negation

A schema negation $\neg S$ is a schema which contains all the bindings of the same signature as those of the schema S but which are not contained in S.

Abstract Syntax A schema negation is composed of a schema

$$\mathsf{SNEGATION} = \neg \mathsf{SCHEMA}$$

Representation and transformation

Production	Concrete	Abstract
'$\neg$ ',LogSch4	$\neg S$	$\neg [\![S]\!]^{S}$

Type The signature of a negated schema $\neg S$ is the same signature as that of the schema S:

$$(\!| \neg\ S\ |\!)^{\mathcal{T}s} = (\!| S\ |\!)^{\mathcal{T}s}.$$

Meaning The bindings of a negated schema $\neg S$ are those bindings which have the same signature as S but are not bindings of S:

$$(\!| \neg\ S\ |\!)^{\mathcal{M}s} = (\!| S\ |\!)^{\mathcal{M}\mathcal{T}s} \setminus (\!| S\ |\!)^{\mathcal{M}s}.$$

Note: This is simpler than in (Spivey, 1988), where this complement had to be combined with the global part of the environment. This was necessary in the original semantics, because the meaning of a schema involved not only the components of the schema, but also the global variables to which the schema might refer.

9.5 Schema Disjunction

The schema disjunction $S_1 \vee S_2$ is a schema whose signature is the join of the signatures of the two schemas S_1 and S_2 and whose property is the disjunction of the two schemas' properties.

Abstract Syntax A schema disjunction is composed of two schemas.

SDISJUNCTION = SCHEMA ∨ SCHEMA

Representation and transformation

Production	Concrete	Abstract
LogSch2, '∨' ,LogSch3	$S_1 \vee S_2$	$[S_1]^S \vee [S_2]^S$

Type The signature of a schema disjuinction $S_1 \vee S_2$ is the join of the two schemas S_1 and S_2 :

$$(\!| S_1 \vee S_2 |\!)^{T_S} = \langle (\!| S_1 |\!)^{T_S}, (\!| S_2 |\!)^{T_S} \rangle \; ; \cup.$$

Note: The schema disjunction $S_1 \vee S_2$ is well-typed only if the signature of the two schemas S_1 and S_2 are type compatible.

Meaning The bindings of a disjoined schema are all those with its signature which are extensions of bindings in one or other of the operand schemas:

$$(\!| S_1 \vee S_2 |\!)^{M_S} = (\langle (\!| S_1 |\!)^{M T_S}, (\!| S_2 |\!)^{M_S} \rangle \cup \langle (\!| S_1 |\!)^{M_S}, (\!| S_2 |\!)^{M T_S} \rangle) \; ; \cup.$$

9.6 Schema Conjunction

Abstract Syntax A schema conjunction is composed of two schemas

SCONJUNCTION = SCHEMA $\wedge$ SCHEMA

Representation and transformation

Production	Concrete	Abstract
LogSch3, '$\wedge$' ,LogSch4	$S_1 \wedge S_2$	$[\![S_1]\!]^S \wedge [\![S_2]\!]^S$

Type The signature of a schema conjunction $S_1 \wedge S_2$ is the join of the two schemas S_1 and S_2 :

$$(\![S_1 \wedge S_2]\!)^{\mathcal{T}s} \quad = \quad ((\![S_1]\!)^{\mathcal{T}s}, (\![S_2]\!)^{\mathcal{T}s}) \,;\, \sqcup.$$

Note: The schema conjunction $S_1 \wedge S_2$ is well-typed only if the two schemas S_1 and S_2 are well-typed and their signatures are type compatible.

Meaning The bindings of a conjoined schema are all those with its signature which are extensions of bindings in both of the operand schemas:

$$(\![S_1 \wedge S_2]\!)^{\mathcal{M}s} \quad = \quad ((\![S_1]\!)^{\mathcal{M}s}, (\![S_2]\!)^{\mathcal{M}s}) \,;\, \sqcup.$$

Note: Spivey (1988) has already remarked on the similarity with the semantics of the parallel composition operator in the traces model of CSP.

9.7 Schema Implication

Abstract Syntax A schema implication is composed of two schemas.

SIMPLICATION = SCHEMA $\Rightarrow$ SCHEMA

Production	Concrete	Abstract
LogSch2, '$\Rightarrow$' ,LogSch1	$S_1 \Rightarrow S_2$	$[S_1]^S \Rightarrow [S_2]^S$

Type The signature of a schema implication $S_1 \Rightarrow S_2$ is the join of the two schemas S_1 and S_2 :

$$\llparenthesis S_1 \Rightarrow S_2 \rrparenthesis^{Ts} = (\llparenthesis S_1 \rrparenthesis^{Ts}, \llparenthesis S_2 \rrparenthesis^{Ts}) \mathbin{;} \sqcup.$$

Note: The schema implication $S_1 \Rightarrow S_2$ is well-typed only if the two schemas S_1 and S_2 are well-typed and their signatures are type compatible.

Meaning The meaning of the schema implication $S_1 \Rightarrow S_2$ is the same as the meaning of the schema disjunction $\neg S_1 \vee S_2$:

$$\llparenthesis S_1 \Rightarrow S_2 \rrparenthesis^{Ms} = \llparenthesis \neg S_1 \vee S_2 \rrparenthesis^{Ms}.$$

9.8 Schema Equivalence

Abstract Syntax A schema equivalence is composed of two schemas.

SEQUIVALENCE = SCHEMA $\Leftrightarrow$ SCHEMA

Representation and transformation

Production	Concrete	Abstract
LogSch, '$\Leftrightarrow$' ,LogSch1	$S_1 \Leftrightarrow S_2$	$[S_1]^S \Leftrightarrow [S_2]^S$

Type The signature of a schema equivalence $S_1 \Leftrightarrow S_2$ is the join of the two schemas S_1 and S_2 :

$$(\!| S_1 \Leftrightarrow S_2 |\!)^{\mathcal{T}s} = \langle (\!| S_1 |\!)^{\mathcal{T}s}, (\!| S_2 |\!)^{\mathcal{T}s} \rangle \mathbin{;} \sqcup.$$

Note: The schema equivalence $S_1 \Leftrightarrow S_2$ is well-typed only if the two schemas S_1 and S_2 are well-typed and their signatures are type compatible.

Meaning The bindings are all those with this signature which are extensions of bindings in neither or both of the operand schema expressions:

$$(\!| S_1 \Leftrightarrow S_2 |\!)^{\mathcal{M}s} = (\!| S_1 \Rightarrow S_2 \wedge S_2 \Rightarrow S_1 |\!)^{\mathcal{M}s}.$$

9.9 Schema Projection

The schema projection operator ($\upharpoonright$) hides all the components of its first argument except those which are also components of its second argument.

Abstract Syntax A schema projection is composed of two schemas.

$$\mathsf{SPROJECTION} = \mathsf{SCHEMA} \upharpoonright \mathsf{SCHEMA}$$

Representation and transformation

Production	Concrete	Abstract
CmpndSch2, '$\upharpoonright$' ,LogSch	$S \upharpoonright T$	$[S]^S \upharpoonright [T]^S$

Type The signature of a projection $S_1 \upharpoonright S_2$ includes those names in both the domains of the signatures of S_1 and S_2. The type given to each such name is taken from S_1. Note that if names are given types by both S_1 and S_2 those types must be the same (that is, the signatures must be consistent):

$$\llparenthesis S_1 \upharpoonright S_2 \rrparenthesis^{\mathcal{T}s} = (\llparenthesis S_1 \rrparenthesis^{\mathcal{T}s}, \llparenthesis S_2 \rrparenthesis^{\mathcal{T}s}) \,;\, \sqcap$$

Meaning The value of the projection $S_1 \upharpoonright S_2$ is the set of bindings which satisfy S_1, restricted to the alphabet of S_2:

$$\llparenthesis S_1 \upharpoonright S_2 \rrparenthesis^{\mathcal{M}s} = (\llparenthesis S_1 \rrparenthesis^{\mathcal{M}s}, \llparenthesis S_2 \rrparenthesis^{\mathcal{M}\mathcal{T}s}) \,;\, \sqcap.$$

Note: Spivey (1988) gives two forms of projection operator used in a schema expression such as $S_1 \upharpoonright S_2$. The weak operator hides those components of S_1 which are not in the signature of S_2. The strong form requires the components to satisfy the axioms of S_2 as well. We give the semantics for the weak operator.

9.10 Schema Hiding

The hiding operator (\) takes a schema expression as its first operand and an identifier list as its second operand. The result is a schema expression whose components are those of the operand schema excluding those named in the list.

Abstract Syntax A hidden schema is composed of a schema and a list of names.

SHIDING = SCHEMA \ [VARNAME,...,VARNAME]

Representation and transformation

Production	Concrete	Abstract
CmpndSch1, '\' , '(' ,VarNameList, ')'.	$S \setminus (\ n_1, n_2, \ldots, n_m\ )$	$[S]^S \setminus <\ n_1, n_2, \ldots, n_m\ >$

Type The signature of a schema hiding expression is the signature of S with the names from $(n_1, \ldots, n_n)$ removed. Note that $(n_1, \ldots, n_n)$ may contain names not in the signature of se:

$$⦇S \setminus (n_1, \ldots, n_m)⦈^{\mathcal{T}s} \quad = \quad ⦇S⦈^{\mathcal{T}s} \; ; (\{n_1, \ldots, n_m\} ⩤)$$

Meaning The value of the schema S in which the components $(n_1, \ldots, n_n)$ have been hidden is the set of bindings which satisfy S, with those components removed:

$$⦇S \setminus (n_1, \ldots n_m)⦈^{\mathcal{M}s} \quad = \quad ⦇S⦈^{\mathcal{M}s} \; ; (\{n_1, \ldots n_m\} ⩤)$$

Note: If all the variables are hidden the result is a schema with an empty signature.

9.11 Schema Universal Quantification

Abstract Syntax A schema quantification is constructed from a schema text and a schema.

$$\mathsf{SUNIVQUANT} \;=\; \forall \mathsf{SCHEMATEXT} \bullet \mathsf{SCHEMA}$$

Representation and transformation

Production	Concrete	Abstract
'∀' ,SchemaText, '•' ,Schema	$\forall St \bullet S$	$\forall [\![St]\!]^{ST} \bullet [\![S]\!]^{S}$

Type The signature of a universally quantified schema expression $\forall St \bullet S$ is the signature of S with the names from the signature of St removed:

$$(\!|\forall St \bullet S\,|\!)^{Ts} \;=\; \langle (\!| S \,|\!)^{Ts}, (\!|\langle St\rangle\,|\!)^{Ts}\rangle \;;\!-$$

Note: The signature is well-typed only when St and S is are well-typed and their signatures are compatible.

Meaning The value of a universally quantified schema expression $\forall St \bullet S$ is the set of bindings with the defined signature such that, for all bindings of St, the union of the two bindings is an extension of S:

$$(\!|\forall St \bullet S\,|\!)^{Ms} \;=\; (\!|\neg \exists St \bullet \neg S\,|\!)^{Ms}$$

Note: Note that this definition takes advantage of de Morgan's Law.

9.12 Schema Existential Quantification

Abstract Syntax A schema quantification is composed of a schema text and a schema.

$$\mathsf{SEXISTSQUANT} \ = \ \exists\, \mathsf{SCHEMATEXT} \bullet \mathsf{SCHEMA}$$

Representation and transformation

Production	Concrete	Abstract
'∃' ,SchemaText, '•' ,Schema	$\exists\, St \bullet S$	$\exists [St]^{ST} \bullet [S]^{S}$

Type The signature of an existentially quantified schema expression $\exists\, \boldsymbol{St} \bullet \boldsymbol{S}$ is the signature of S with the names from the signature of St removed:

$$\llparenthesis \exists\, \boldsymbol{St} \bullet \boldsymbol{S} \rrparenthesis^{\mathcal{T}s} \ = \ \langle \llparenthesis \boldsymbol{S} \rrparenthesis^{\mathcal{T}s}, \llparenthesis \langle \boldsymbol{St} \rangle \rrparenthesis^{\mathcal{T}s} \rangle \,; \leftarrow .$$

Note: The signature is well-typed only when $\boldsymbol{St}$ and $\boldsymbol{S}$ is are well-typed and their signatures are compatible.

Meaning The value of an existentially quantified schema expression $\exists\, \boldsymbol{St} \bullet \boldsymbol{S}$ is the set of bindings with signature of $\boldsymbol{S}$ less $\boldsymbol{St}$, such that there is a binding of $\boldsymbol{St}$ so that the union of the two bindings is an extension of $\boldsymbol{S}$:

$$\llparenthesis \exists\, \boldsymbol{St} \bullet \boldsymbol{S} \rrparenthesis^{\mathcal{M}s} \ = \ \langle \llparenthesis \boldsymbol{S} \rrparenthesis^{\mathcal{M}s}, \llparenthesis \langle \boldsymbol{St} \rangle \rrparenthesis^{\mathcal{M}s} \,; - .$$

Note: This definition should be contrasted with the analogous expression for predicates ($\exists\, St \bullet p$) where the well-typing of the predicate is decided in the modified environment.

9.13 Schema Unique Existential Quantification

Abstract Syntax A schema quantification is composed of a schema text and a schema.

$\mathsf{SUNIQUEQUANT} \;\;=\;\; \exists_1\, \mathsf{SCHEMATEXT} \bullet \mathsf{SCHEMA}$

Representation and transformation

Production	Concrete	Abstract
'$\exists_1$' ,SchemaText, '$\bullet$' ,Schema	$\exists_1 St \bullet S$	$\exists_1 [St]^{ST} \bullet [S]^{S}$

Type

$\llparenthesis \exists_1\, St \bullet S \,\rrparenthesis^{T_S} \;\;=\;\; \langle \llparenthesis S \,\rrparenthesis^{T_S}, \llparenthesis (St) \,\rrparenthesis^{T_S} \rangle\, ;-$

Note: The signature is well-typed only when St and S is are well-typed and their signatures are compatible.

Meaning The value of an existentially quantified schema expression $\exists_1\, St \bullet S$ is the set of bindings with signature of S less St, such that there exists a unique binding of St so that the union of the two bindings is an extension of S:

$\llparenthesis \exists_1\, St \bullet S \,\rrparenthesis^{\mathcal{M}_S} \;\;=\;\;$ To he defined

9.14 Schema Renaming

The renaming operation $S[new/old]$ substitutes the new variable name for the old in the schema.

Abstract Syntax A schema renaming consists of a schema and a renaming list.

SRENAMING = SCHEMA RENAMELIST

Representation and transformation

Production	Concrete	Abstract
CmpndSch1,RenameList	$S[x_1/y_1, x_2/y_2, \ldots x_n/y_n]$	$[S]^{S} < x_1/y_1, x_2/y_2, \ldots x_n/y_n >$

Type Schema renaming changes the names of the elements in the bindings, and hence the signature.

$$(\!|S[Nl]|\!)^{\mathcal{T}s} = (\!|S|\!)^{\mathcal{T}s} ; \exists(\{Nl\}^{\mathcal{N}} \times 1)$$

Meaning

$$(\!|S[Nl]|\!)^{\mathcal{M}s} = (\!|S|\!)^{\mathcal{M}s} ; \exists(\{Nl\}^{\mathcal{N}} \times 1)$$

Note: When more than one variable is to be substituted, the substitution is simultaneous. Any substitutions for non-existent names are ignored. Each old name can only be substituted by one new name. Likewise, each new name can be a substitute for only one old name.

9.15 Schema Substitution

Abstract Syntax

SCHEMASUBSTITUTION = EXP⊙SCHEMA

Representation and transformation

Production	Concrete	Abstract
Expression,'⊙' ,Schema	$b \odot S$	$[b]^{\mathcal{E}} \odot (S)^{\mathcal{S}}$

Type

$$(\!|b \odot S\,|\!)^{\mathcal{T}s} \;=\; \langle 1, [\![b\,]\!]^{\mathcal{T}} \,;\, schemaT^{-1}\rangle \,;\, \oplus \,;\, (\!|S\,|\!)^{\mathcal{T}s}$$

Meaning

$$(\!|b \odot S\,|\!)^{\mathcal{M}s} \;=\; \langle 1, [\![b\,]\!]^{\mathcal{M}} \,;\, (\_,\_)\rangle \,;\, \oplus \,;\, (\!|S\,|\!)^{\mathcal{M}}$$

9.16 Free Variables

Table 27: Schemas and their free variables and alphabet

Schema	Free Variables	Alphabet
S	$\{S\}$	
$S[x_1, \ldots, x_n]$	$\{S\} \cup \phi_e x_1 \cup \ldots \cup \phi_e x_n$	
$[d \mid p]$	$\phi_d(d \mid p)$	αd
$\neg T$	$\phi_s T$	αT
$(S \land T)$	$\phi_s S \cup \phi_s T$	$\alpha S \cup \alpha T$
$(S \lor T)$	$\phi_s S \cup \phi_s T$	$\alpha S \cup \alpha T$
$(S \Rightarrow T)$	$\phi_s S \cup \phi_s T$	$\alpha S \cup \alpha T$
$(S \Leftrightarrow T)$	$\phi_s S \cup \phi_s T$	$\alpha S \cup \alpha T$
$(\forall St \bullet T)$	$\phi_d St \cup \phi_s T$	$\alpha T \setminus \alpha St$
$(\exists St \bullet T)$	$\phi_d St \cup \phi_s T$	$\alpha T \setminus \alpha St$
$(\exists_1 St \bullet T)$	$\phi_d St \cup \phi_s T$	$\alpha T \setminus \alpha St$

Table 28: Substitution into Schemas

Substitution	Equivalence
$\boldsymbol{b \odot S}$	S
$\boldsymbol{b \odot S[x_1, \ldots, x_n]}$	$S[b \odot x_1, \ldots b \odot x_n]$
$\boldsymbol{b \odot [d \mid p]}$	$[b \odot d \mid (b \setminus [d]) \odot p]$
$\boldsymbol{b \odot \neg T}$	$\neg b \odot T$
$\boldsymbol{b \odot (S \land T)}$	$(b \odot S) \land (b \odot T)$
$\boldsymbol{b \odot (S \lor T)}$	$(b \odot S) \lor (b \odot T)$
$\boldsymbol{b \odot (S \Rightarrow T)}$	$(b \odot S) \Rightarrow (b \odot T)$
$\boldsymbol{b \odot (S \Leftrightarrow T)}$	$(b \odot S) \Leftrightarrow (b \odot T)$
$\boldsymbol{b \odot (\forall St \bullet T)}$	$\forall b \odot St \bullet b \odot T$
$\boldsymbol{b \odot (\exists St \bullet T)}$	$\exists b \odot St \bullet b \odot T$
$\boldsymbol{b \odot (\exists_1 St \bullet T)}$	$\exists_1 b \odot St \bullet b \odot T$

10 Paragraph

$$\begin{aligned}\mathsf{PAR} = {} & \mathsf{GIVENSETDEF}\\ | {} & \mathsf{GLOBALPRED}\\ | {} & \mathsf{GLOBALDECL}\\ | {} & \mathsf{GENERICDECL}\\ | {} & \mathsf{GLOBALDEF}\\ | {} & \mathsf{GENERICDEF}\\ | {} & \mathsf{CONJECTURE}\end{aligned}$$

Each paragraph of Z can do two things: Augment the environment by a declaration and strengthen the property by a predicate. Each paragraph is considered as a relation between environments. The domain of this relation contains all the environments in which the paragraph is well-typed and any predicates contained within it are true. These environments are related to those which include the new variables declared in their signature and which satisfy any property denoted by the paragraph.

$$(\mathsf{PAR})^{\mathcal{T}} : \mathit{Tenv} \twoheadrightarrow \mathit{Tenv}$$

$$(\mathsf{PAR})^{\mathcal{M}} : \mathit{Env} \leftrightarrow \mathit{Env}$$

We can prove the following

$$\vdash (\boldsymbol{Par})^{\mathcal{M}} ; \Upsilon \subseteq \Upsilon ; (\boldsymbol{Par})^{\mathcal{T}}$$

10.1 Given Sets

The given set definition $[\ X_1, X_2, \ldots, X_n\ ]$ introduces the sets $X_1, X_2, \ldots, X_n$ without determining their elements.

Note: Distinctly named given sets have distinct types and hence are incomparable.

Abstract Syntax

$$\mathsf{GIVENSETDEF} \ = \ \mathbf{given}\ [\mathsf{WORD}, \mathsf{WORD}, \ldots, \mathsf{WORD}]$$

Representation and transformation

Production	Concrete	Abstract
'[' , Word,{',',Word} , ']'	$[\ X_1, X_2, \ldots, X_n\ ]$	$\mathbf{given}\ (X_1, \ldots, X_n)$

Type The declaration of given sets **given** $[x_1, \ldots, x_n]$ causes the type environment to be suitably enriched. Each name is given the power set type of the given type of that name. These declarations override the environment. Note that a given set definition of N results in N having the type *powerT givenT* N.

$$\{\mathbf{given}(X_1, \ldots, X_n)\ \}^{\mathcal{T}} \ = \ \langle 1, (\{X_1, \ldots, X_n\} \lhd givenT \mathbin{;} powerT)^{\circ}\rangle \mathbin{;} \oplus$$

Meaning To enrich the meaning environment, we construct a binding of the given set names (those in ran s) to typed values in the world of sets—for this to be correct, the bindings must be such that the given sets do indeed have power set type. The environment is updated with this binding.

$$\{\mathbf{given}(X_1, \ldots, X_n)\ \}^{\mathcal{M}} \ = \ \langle 1, (\{X_1, \ldots, X_n\} \lhd givenT \mathbin{;} (powerT, Carrier))^{\circ}\rangle \mathbin{;} \oplus$$

10.2 Constraints

A Constraint is a predicate appearing on its own as a paragraph. It denotes a property of the values of variables declared elsewhere with global scope. This property is conjoined to the global property.

Abstract Syntax

GLOBALPRED = **where** PRED

Representation and transformation

Production	Concrete	Abstract
Predicate	P	where$[P]^{\mathcal{P}}$

Type A constraint adds nothing to the environment, so it is that subset of the identity relation restricted to the environments in which the predicate is true.

For the type environment:

$$\{P\}^{\mathcal{T}} = {}^{1}\llbracket P \rrbracket^{\mathcal{T}}$$

Meaning For meaning environment:

$$\{P\}^{\mathcal{M}} = {}^{1}\llbracket P \rrbracket^{\mathcal{M}}$$

10.3 Global Declaration

An axiomatic definition introduces variables and specifies further properties of the elements denoted by them.

Abstract Syntax

GLOBALDECL = **defn** SCHEMATEXT

Representation and transformation

Prodnction	Concrete	Abstract
'<u>AX</u>' ,DeclPart, '<u>ST</u>',AxiomPart, '<u>END</u>'	'<u>AX</u>' D '<u>ST</u>' P'<u>END</u>'	**defn** $[D]^{\mathcal{D}} \mid [P]^{\mathcal{P}}$
'<u>AX</u>' ,DeclPart, '<u>END</u>'	'<u>AX</u>' D '<u>END</u>'	**defn** $[D]^{\mathcal{D}} \mid$ **true**

The abstract form of an axiomatic definition is a pair of paragraphs, one containing a declaration and the other a predicate. If the AxiomPart is omitted the the abstract form is one declaration paragraph.

Type When new variables are declared the environment is enriched by a function from their names to one from their empty generic parameter list to their meaning. We give as its value a set of bindings, one for each name declared. In obtaining the binding, we enrich the environment with the declaration in such a way that the constraint is satisfied. The names in the declaration are bound to their values in this enriched environment. Formally:

$$(\mathbf{defn} D \mid P)^{\mathcal{T}} = (D \mid P)^{\mathcal{T}}$$

Meaning

$$(\mathbf{defn} D \mid P)^{\mathcal{M}} = (D \mid P)^{\mathcal{M}}$$

Note *The sets from which the elements denoted by the variables can be drawn are defined by the conjunction of the constraint of the* DeclPart *and the property in the* AxiomPart.

The signature of the DeclPart *is joined to the global signature. The constraint in the* DeclPart *and the property of the* AxiomPart *are conjoined to the global property.*

10.4 Generic Declarations

A generic definition of variables adds these variables to the dictionary and maps them to a function from all possible instantiations of their generic parameters to the values of the variables with these instantiations.

Abstract Syntax

GENERICDECL = **gendef** [WORD, WORD, ..., WORD] const SCHEMATEXT

Representation and transformation

Production	Concrete	Abstract
'GEN',GenFormals,'BAR', DeclPart,'ST', AxiomPart,'END'	'GEN' [$X_1, X_2, \ldots X_n$]'BAR' D 'ST' P'END'	gendef ($X_1, X_2, \ldots, X_n$) const$[D]^{\mathcal{D}}$ where $[P]^{\mathcal{P}}$
'GEN',GenFormals,'BAR', DeclPart,'END'	'GEN' [$X_1, X_2, \ldots, X_n$]'BAR' D 'END'	gendef ($X_1, X_2, \ldots, X_n$) const$[D]^{\mathcal{D}}$ where true

Type

Value A generic definition introduces a family of variables, parameterised by the generic parameters of the list GenFormals.

Note *In a* GenericDef, *the* DeclPart *declares the names of the generic variables whose types can be determined upon instantiation of the formal parameters. The predicate in the* AxiomPart *determines the elements denoted by the variables for each value of the formal parameters.*

Recursive generic definitions are not allowed. The generic definition must not place any restriction on the generic parameters.

A generic variable has global scope, excluding the declaration list in which it is declared and any construct in which its name is re-used for a local variable.

The parameters of a generic definition are local to the definition, but they can be instantiated by elements of set type when the generic variable is used.

A generic definition does not give a single type: rather, a function from the generic parameters to types is defined.

10.5 Global Definitions

Abstract Syntax

GLOBALDEF = **abbr** WORD $\widehat{=}$ EXP

Representation and transformation

Note: A SchemaDefdefines a new *schema*. There are two forms for a schema definition. The horizontal is the primary form. The vertical form, using a schema box, is given a meaning in terms of an equivalent horizontal definition.

Production	Concrete	Abstract
SchemaName, '$\widehat{=}$' ,Schema		
'<u>SCH</u>' , SchemaName, '<u>IS</u>' ,DeclPart, '<u>ST</u>',AxiomPart, '<u>END</u>'		
'<u>SCH</u>' , SchemaName, '<u>IS</u>' ,DeclPart, '<u>END</u>'		
Ident, '==' ,Expression		

Type When a schema or variable is declared the name is added to the type-environment and is mapped to the type of the schema or expression.

$$\{\!\!\{\mathbf{abbr} N \mathrel{\widehat{=}} X\}\!\!\}^{\mathcal{T}} = \langle 1, (N^{\circ}, [\![X]\!]^{\mathcal{T}}) \,;\, \{-\}\rangle \,;\, \oplus$$

Meaning When a schema or variable is declared the name of the schema is added to the environment and is mapped to the meaning of the schema or expression.

$$\{\!\!\{\mathbf{abbr} N \mathrel{\widehat{=}} X\}\!\!\}^{\mathcal{M}} = \langle 1, (N^{\circ}, [\![X]\!]^{\mathcal{M}}) \,;\, \{-\}\rangle \,;\, \oplus$$

Note

- *The horizontal form of the definition defines the schema with name* SchemaName *as the schema denoted by the* SchemaExpr.
- *The vertical form of the definition defines the schema with name* SchemaName *as the schema denoted by the schema expression constructed from the schema text comprising the horizontal equivalents of the* DeclPart *and the* AxiomPart *(see Vertical Form).*

A SchemaName *may be used to define only one schema within a specification.*

A Schema has global scope except within the text of its definition. Recursive schema definitions are not allowed. The scope of variables introduced in the DeclPart *is local to the* SchemaDef *and includes the* AxiomPart.

10.6 Generic Definitions

A generic definition of variables adds these variables to the environment and maps them to a fuuction from all possible instantiations of their generic parameters to the values of the variables with these instantiations.

Abstract Syntax

GENERICDEF = **abbr** WORD[WORD, WORD, ..., WORD] $\mathrel{\widehat=}$ EXP

Representation and transformation

Production	Concrete	Abstract
SchemaName,GenFormals, '$\mathrel{\widehat=}$' ,Schema		
'<u>SCH</u>' , SchemaName,GenFormals, '<u>IS</u>' ,DeclPart, '<u>ST</u>',AxiomPart, '<u>END</u>'		
'<u>SCH</u>' , SchemaName,GenFormals, '<u>IS</u>' ,DeclPart, '<u>END</u>'		
Ident,GenFormals, '==' ,Expression ;		
Word, InGen, Word, '==' ,Expression		
PreGeu, Word, '==' ,Expression		

Type

$$
\begin{array}{l}
\{\mathbf{abbr}N[S_1, \ldots, Sm] \mathrel{\widehat=} X\}^{\mathcal{M}} \;=\\
\quad (\\
\qquad 1,\\
\qquad {}^{\wedge}(\langle 1, \langle\langle S_1{}^{\circ}, Ptype^{\circ}\, ;\ni\rangle, \ldots, \langle S_m{}^{\circ}, Ptype^{\circ}\, ;\ni\rangle\rangle\, ;\{\ldots\}\rangle)\, ;{}^{\exists}(\langle\langle [\![S_1]\!]^{T}, \ldots, [\![S_m]\!]^{T}\rangle, [\![X]\!]^{T}\rangle)\\
\quad)\, ;\oplus
\end{array}
$$

Value

Note *In a* GenericDef, *the* DeclPart *declares the names of the generic variables whose types can be determined upon instantiation of the formal parameters.*

An abbreviation definition can be used to define a possibly generic variable which is named by an identifier Abbrev.

The variable defined by the expression can take three forms:

- *Possibly Generic Variable* Ident.
- *Prefix Generic Symbol* PreGen.

- *Infix Generic Symbol* InGen.

  In the latter two cases, the names of the generic parameters. Word *indicate the positions of the actual parameters which can be supplied when the variables are used.*

  A schema may be defined with generic parameters and when used it must be always instantiated.

11 Specification

A specification is constructed from a seqnence of paragraphs:

Abstract Syntax

SPEC = PAR ,..., PAR

Representation and transformation

Production	Concrete	Abstract
[Paragraph] , {Narrative,Paragraph}, [Narrative]	P_1 Narrative... Narrative P_n	$[P_1]^{PAR}$ **and** ...**and** $[P_n]^{PAR}$

Type A specification is well-typed if the empty type environment is in the domain of the typing relation.

Meaning The meaning of a specification is the set of environments which are related to the empty environment by the paragraphs of the text. These are all the environments which are enrichments of the empty environment by the specification. A sequence of paragraphs can be composed together. They denote a relation between environments. This relation is the sequential composition of the relations denoted by the individual paragraphs.

$$zmnP_1\mathbf{and}\ldots\mathbf{and}P_n \;=\; {}^{\wedge}((P_1)^{\mathcal{M}};\ldots;(P_n)^{\mathcal{M}})\varnothing$$

Note *A Z specification consists of a sequence of paragraphs separated by paragraph separators. These paragraph separators may include explanatory text. The global signature and property are constructed from the meanings of these paragraphs.*

A paragraph is either a definition or a constraint.

A definition introduces Basic types, schemas, or variables (named elements, sets tuples or bindings) together with constraints on them. The effect of a definition is to augment the global signature and to conjoin its constraint with the global property.

A constraint denotes a property on variables and schemas declared elsewhere. The effect of a constraint is to conjoin its property with the global property.

A specification is well typed if every term and predicate within the paragraphs is well typed.

A Abstract Syntax

This annex contains the abstract syntax for Z. The metalanguage used is a form of BNF. The notation $X, \ldots, X$ denotes zero or more occurrences of X separated by commas.

A.1 Specification

```
SPEC = PAR ,..., PAR
```

A.2 Paragraph

```
PAR         = GIVENSETDEF
            | GLOBALPRED
            | GLOBALDECL
            | GENERICDECL
            | GLOBALDEF
            | GENERICDEF
            | CONJECTURE

GIVENSETDEF = given [WORD,WORD,.. ,WORD]
GLOBALPRED  = where PRED
GLOBALDECL  = defn SCHEMATEXT
GENERICDECL = gendef [WORD,WORD,...,WORD] const SCHEMATEXT
GLOBALDEF   = abbr WORD ≙ EXP
GENERICDEF  = abbr WORD[WORD,WORD,...,WORD] ≙ EXP
CONJECTURE  = conj DECL | PRED,...,PRED ⊢ PRED,...,PRED
```

A.3 Schema

SCHEMA	=	SDES
	|	GENSDES
	|	SCONSTRUCTION
	|	SNEGATION
	|	SDISJUNCTION
	|	SCONJUNCTION
	|	SIMPLICATION
	|	SEQUIVALENCE
	|	SPROJECTION
	|	SHIDING
	|	SUNIVQUANT
	|	SEXISTSQUANT
	|	SUNIQUEQUANT
	|	SRENAMING
	|	SCOMPOSITION
	|	SDECORATION
	|	SCHEMASUBSTITUTION

SDES	=	WORD
SCONSTRUCTION	=	(DECL | PRED)
SNEGATION	=	¬SCHEMA
SDISJUNCTION	=	SCHEMA ∨ SCHEMA
SCONJUNCTION	=	SCHEMA ∧ SCHEMA
SIMPLICATION	=	SCHEMA ⇒ SCHEMA
SEQUIVALENCE	=	SCHEMA ⇔ SCHEMA
SPROJECTION	=	SCHEMA ↾ SCHEMA
SHIDING	=	SCHEMA \ [VARNAME, . . . , VARNAME]
SUNIVQUANT	=	∀SCHEMATEXT • SCHEMA
SEXISTSQUANT	=	∃SCHEMATEXT • SCHEMA
SUNIQUEQUANT	=	$\exists_1$ SCHEMATEXT • SCHEMA
SRENAMING	=	SCHEMA RENAMELIST
SCOMPOSITION	=	SCHEMA ; SCHEMA
SDECORATION	=	SCHEMA DECOR
SCHEMASUBSTITUTION	=	EXP⊙SCHEMA

A.4 Schema Text

```
SCHEMATEXT        = SIMPLESCT
                  | CMPNDSCT
                  | SCTSUBSTITUTION

SIMPLESCT         = DECL

CMPNDSCT          = DECL | PRED

SCTSUBSTITUTION   = EXP⊙SCHEMATEXT
```

A.5 Declaration

```
DECL              = SIMPLEDECL
                  | SCHEMAINCL
                  | COMPNDECL
                  | DECLSUBSTITUTION

SIMPLEDECL        = VARNAME, VARNAME, ..., VARNAME : EXP

SCHEMAINCL        = SCHEMA

COMPNDECL         = DECL; DECL

DECLSUBSTITUTION  = EXP⊙DECL
```

A.6 Predicate

```
PRED              = EQUALITY
                  | MEMBERSHIP
                  | TRUTH
                  | FALSEHOOD
                  | NEGATION
                  | DISJUNCTION
                  | CONJUNCTION
                  | IMPLICATION
                  | EQUIVALENCE
                  | UNIVERSALQUANT
                  | EXISTSQUANT
                  | UNIQUEQUANT
                  | SCHEMAPRED
                  | PREDSUBSTITUTION

EQUALITY          = EXP  =  EXP

MEMBERSHIP        = EXP  ∈  EXP
```

TRUTH	=	**true**
FALSEHOOD	=	**false**
NEGATION	=	¬PRED
DISJUNCTION	=	PRED ∨ PRED
CONJUNCTION	=	PRED ∧ PRED
IMPLICATION	=	PRED ⇒ PRED
EQUIVALENCE	=	PRED ⇔ PRED
UNIVERSALQUANT	=	∀SCHEMATEXT • PRED
EXISTSQUANT	=	∃SCHEMATEXT • PRED
UNIQUEQUANT	=	$\exists_1$ SCHEMATEXT • PRED
SCHEMAPRED	=	SCHEMA
PREDSUBSTITUTION	=	EXP⊙PRED

A.7 Expression

EXP	=	IDENT
	|	GENINST
	|	NUMBERL
	|	STRINGL
	|	SETEXTN
	|	SETCOMP
	|	POWERSET
	|	TUPLE
	|	PRODUCT
	|	TUPLESELECTION
	|	BINDINGEXTN
	|	THETAEXP
	|	SCHEMAEXP
	|	BINDSELECTION
	|	FUNCTAPP
	|	DEFNDESCR
	|	IFTHENELSE
	|	EXPSUBSTITUTION

IDENT	=	VARNAME
GENINST	=	VARNAME [EXP, EXP, ..., EXP]
NUMBERL	=	NUMBER
STRINGL	=	STRING
SETEXTN	=	{EXP, EXP, ..., EXP}

SETCOMP	=	{SCHEMATEXT • EXP}
POWERSET	=	**P** EXP
TUPLE	=	(EXP, EXP, . . . , EXP, EXP)
PRODUCT	=	EXP × EXP × . . . × EXP × EXP
BINDINGEXTN	=	⦇ VARNAME ⇝ EXP, . . . , VARNAME ⇝ EXP⦈
THETAEXP	=	*θ* SCHEMA DECOR
	|	*θ* SCHEMA
BINDSELECTION	=	EXP . VARNAME
FUNCTAPP	=	EXP(EXP)
DEFNDESCR	=	*μ* SCHEMATEXT • EXP
SCHEMAEXP	=	SCHEMA
EXPSUBSTITUTION	=	EXP ⊙ EXP

A.8 Identifier

VARNAME	=	WORD DECOR
DECOR	=	[STK, . . . , STK]
RENAMELIST	=	[VARNAME/VARNAME, . . . , VARNAME/VARNAME]

B Representation Syntax

The concrete representation for Z is defined in four parts. The first is a context-free grammar, which conforms to the BSI standard for grammars. The second, lexical analysis, describes the rules according to which the character sequences are grouped into tokens. The Character set describes the character set required to represent a Z specification. The fourth section, graphical conventions, details the conventions used for layout that are adopted in this standard.

B.1 Grammar

The grammar is described using a BNF notation which employs the following special symbols:

,	the concatenate symbol
=	the define symbol
|	the definition separator symbol
[ ]	enclose optional syntactic items
{ }	enclose syntactic items which may occur zero or more times
' '	single quotes used to enclose terminal symbols
MetaIdentifier	non-terminal symbols written in sans-serif font.
;	terminator symbol denoting the end of a rule
−	subtraction from a set of terminals.
? ...?	"User defined rule.

The concatenate symbol has a higher precedence than the definition separator symbol.

B.1.1 Specification

```
Specification = [ Paragraph] ,{Narrative,Paragraph},[ Narrative] ;

Paragraph     = GivenSetDef
              | StructuredSetDef
              | AxiomaticDef
              | Constraint
              | GenericDef
              | AbbreviationDef
              | SchemaDef
              | Conjecture;
```

B.1.2 Given Set

```
GivenSetDef = '[',Word,{',',Word},']';
```

B.1.3 Structured Set

```
StructuredSetDef = Word,'::=',Branch,{'|',Branch};
```

```
Branch          = Word
                | Ident,'⟪',Expression,'⟫';
```

B.1.4 Global Definition

```
AxiomaticDef = 'AX',DeclPart,'END'
             | 'AX',DeclPart,'ST',AxiomPart,'END';


Constraint   = Predicate;
```

B.1.5 Generic Definition

```
GenericDef      = 'GEN',GenFormals,'BAR',DeclPart,'END'
                | 'GEN',GenFormals,'BAR',DeclPart,'ST',AxiomPart,'END';


AbbreviationDef = VarAbbrev
                | PreGenAbbrev
                | InGenAbbrev;

VarAbbrev       = Ident,'==',Expression
                | Ident,GenFormals,'==',Expression;;

PreGenAbbrev    = PreGen,Word,'==',Expression;

InGenAbbrev     = Word,InGen,Word,'==',Expression;
```

B.1.6 Schema Definition

```
SchemaDef = SchemaName,'≙',Schema
          | SchemaName,GenFormals,'≙',Schema
          | 'SCH',SchemaName,'IS',DeclPart,'ST',AxiomPart,'END'
          | 'SCH',SchemaName,GenFormals,'IS',DeclPart,'ST',AxiomPart,'END'
          | 'SCH',SchemaName,'IS',DeclPart,'END'
          | 'SCH',SchemaName,GenFormals,'IS',DeclPart,'END';
```

B.1.7 Declaration

```
DeclPart        = Declaration,{Nl,Declaration};

Declaration     = BasicDecl
                | CompoundDecl
                | DeclSubstitution;

CompoundDecl    = BasicDecl,'; ',BasicDecl,{'; ',BasicDecl};
```

```
BasicDecl        = SimpleDecl
                 | SchemaIncl;

SimpleDecl       = DeclName,{',',DeclName},':',Expression;

SchemaIncl       = Schema;

DeclSubstitution = Expression,'⊙',Declaration;
```

B.1.8 Schema Text

```
SchemaText      = CmpndSctext
                | SimpleSctext
                | SctSubstitution;

CmpndSctext     = Declaration,'|',Predicate;

SimpleSctext    = Declaration;

SctSubstitution = Expression,'⊙',SchemaText;
```

B.1.9 Schema

```
Schema          = SUnivQuant
                | SExistsQuant
                | SUniqueQuant
                | LogSch;

LogSch          = SEquivalence
                | LogSch1;

LogSch1         = SImplication
                | LogSch2;

LogSch2         = SDisjunction
                | LogSch3;

LogSch3         = SConjunction
                | LogSch4;

LogSch4         = SNegation
                | CmpndSch;

CmpndSch        = SComposition
                | CmpndSch1;
```

B REPRESENTATION SYNTAX

```
CmpndSch1          =  SRenaming
                   |  SHiding
                   |  CmpndSch2;

CmpndSch2          =  SProjection
                   |  CmpndSch3;

CmpndSch3          =  PreSchema
                   |  CmpndSch4;

CmpndSch4          =  SDecoration
                   |  BasicSch;

BasicSch           =  SConstruction
                   |  SchemaRef
                   |  GenSchemaRef
                   |  SchemaSubstitution
                   |  '(',Schema,')';

SUnivQuant         =  '∀',SchemaText,'•',Schema;

SExistsQuant       =  '∃',SchemaText,'•',Schema;

SUniqueQuant       =  '∃₁',SchemaText,'•',Schema;

SEquivalence       =  LogSch,'⇔',LogSch1;

SImplication       =  LogSch2,'⇒',LogSch1;

SDisjunction       =  LogSch2,'∨',LogSch3;

SConjunction       =  LogSch3,'∧',LogSch4;

SNegation          =  '¬ ',LogSch4;

SComposition       =  CmpndSch,';',CmpndSch1;

SHiding            =  CmpndSch1,'\','(',VarNameList,')';

SRenaming          =  CmpndSch1,RenameList;

SProjection        =  CmpndSch2,'↾',LogSch;

PreSchema          =  'pre ',CmpndSch3;

SDecoration        =  Schema,Decoration;
```

```
SConstruction      = '[',Declaration,'|',Predicate.']'
                   | '[',Declaration,']';

SchemaRef          = SchemaName;

GenSchemaRef       = SchemaName,'[',Expression,{',',Expression}']';

SchemaSubstitution = Expression,'⊙',Schema;
```

B.1.10 Predicate

```
AxiomPart     = Predicate,{Sep,Predicate};

Sep           = ';'
              | Nl;

Predicate     = UnivQuant
              | ExistsQuant
              | UniqueQuant
              | LogPred;

LogPred       = Equivalence
              | LogPred1;

LogPred1      = Implication
              | LogPred2;

LogPred2      = Disjunction
              | LogPred3;

LogPred3      = Conjunction
              | BasicPred;

BasicPred     = PreRelPred
              | CmpndRelPred
              | SchemaPred  ~  '(',Schema,')'
              | Truth
              | Falsehood
              | '(',Predicate,')'
              | Negation
              | Membership
              | Equality
              | InRelPred
              | PredSubstitution;

UnivQuant     = '∀',SchemaText,'•',Predicate;
```

```
ExistsQuant      = ‘∃’,SchemaText,‘•’,Predicate;

UniqueQuant      = ‘∃_1’,SchemaText,‘•’,Predicate;

Equivalence      = LogPred,‘⇔’,LogPred1;

Implication      = LogPred2,‘⇒’,LogPred1;

Disjunction      = LogPred2,‘∨’,LogPred3;

Conjunction      = LogPred3,‘∧’,BasicPred;

Negation         = ‘¬ ’,BasicPred;

InRelPred        = Expression,InRel,Expression;

CmpndRelPred     = InRelPred ,Rel,Expression,{Rel,Expression};

Rel              = ‘∈’
                 | ‘=’
                 | InRel;

PreRelPred       = PreRel,Expression;

SchemaPred       = CmpndSch;

Truth            = ‘true’;

Falsehood        = ‘false’;

Membership       = Expression,‘∈’,Expression;

Equality         = Expression,‘=’,Expression;

PredSubstitution = Expression,‘⊙’,Predicate;
```

B.1.11 Expression

```
Expression0      = DefnDescr
                 | Expression;

Expression       = InGenExp
                 | Expression1;

Expression1      = CartProduct
                 | Expression2;
```

```
Expression2     = InFunExp
                | Expression3;

Expression3     = PowerSet
                | PreGenExp
                | Expression4;

Expression4     = FunctApp
                | Expression5;

Expression5     = PostFunExp
                | SuperScript
                | BindSelection
                | TupleSelection
                | Ident
                | GenInstant
                | SchemaExp
                | SetExtn
                | Tuple
                | Sequence
                | Bag
                | BindingExtn
                | ThetaExp
                | SetComp - '{',SchemaExp,'}'
                | LambdaExp
                | NumberI
                | StringI
                | IfThenElse
                | ExpSubstitution
                | '(',Expression0,')';

InGenExp        = Expression1Expression1,InGen,Expression;

CartProduct     = Expression2,'×',Expression2,{'×',Expression2};

InFunExp        = Expression2,InFun,Expression3;

PowerSet        = 'P',Expression5;

PreGenExp       = PreGen,Expression5;

FunctApp        = Expression4,Expression5;

PostFunExp      = Expression5,PostFun,;

SuperScript     = Expression5,^Expression0;

BindSelection   = Expression5,'.',VarName;
```

```
TupleSelection  = Expression5,'.',NumberI;

Ident           = VarName;

GenInstant      = VarName,'[',Expression,{',',Expression}']';

SchemaExp       = Schema;

SetExtn         = '{',Expression0,{',',Expression0},'}';

Tuple           = '(',Expression0,',',Expression0,{',',Expression0},')';

Sequence        = '⟨',Expression0,{',',Expression0},'⟩';

Bag             = '⟦',Expression0,{',',Expression0},'⟧';

BindingExtn     = '⦉ ',VarName,'⇝',Expression0,{',',VarName,'⇝',Expression0},'⦊ ';

ThetaExp        = 'θ',BasicSch,Decoration
                | 'θ',BasicSch;

SetComp         = '{',SchemaText,'•',Expression0,'}'
                | '{',SchemaText,'}';

LambdaExp       = 'λ',SchemaText,'•',Expression;

DefnDescr       = 'μ',SchemaText,'•',Expression
                | 'μ',SchemaText;

NumberI         = Number;

StringI         = String;

IfThenElse      = 'If',Predicate,'Then',Expression ,'Else',Expression,'Fi';

ExpSubstitution = Expression,'⊙',Expression;
```

B.2 Lexical Analysis

Token A *token* is a sequence of characters, as defined in section B.3, conforming to the grammar given in this section, whose terminal symbols are the sets of characters defined in section B.3, and whose sentence symbol is *Token*. The different sorts of token correspond to the sorts of terminal symbols of the grammar of Z, together with an extra sort of space tokens.

A sequence of characters is interpreted as a sequence of non-space tokens by a left-to-right scan taking tokens which are as long as possible and then discarding any *Space* tokens. If it is not possible to do this then the sequence of characters is erroneous.

Note: The text of a Z document in the concrete representation may be considered at three levels: as marks on paper, as a sequence of characters and as a sequence of tokens. The transformation from characters to tokens is given by the following rules; these use the same notation as the syntax definition but differ in meaning in that no two separators may appear between adjacent terminals. Where ambiguity is otherwise possible, two consecutive tokens must be separated by a separator.

```
Token = Word
      | Decoration
      | Narrative
      | Number
      | String
      | Punctuation
      | Space;
```

Operation Names

```
Opname = ‘ _ ’,InFun,[ Dec ],‘ _ ’
       | ‘ _ ’,InGen,‘ _ ’
       | ‘ _ ’,InRel,[ Dec ],‘ _ ’
       | PreGen,‘ _ ’
       | PreRel,[ Dec ],‘ _ ’
       | ‘ _ ’,PostFun,[ Dec ]
       | ‘ _ ’,‘⦇’,‘ _ ’,‘⦈’
       | ‘~’;
```

Variable Names

```
VarName = Name
        | ‘(’,Opname,‘)’;
```

Declaration Names

```
DeclName = Name
         | Opname,;
```

Schema Names

```
SchemaName = Word;
```

Name A name is a decorated word:

```
Name = Word,[ Dec ];
```

B REPRESENTATION SYNTAX

Word There are three sorts of Word:

```
Word          = Alphanumeric
              | Greek
              | Symbolic;

Alphanumeric  = (Letter, {Letter | Digit | ('_', (Letter | Digit))}), {Subscript};

Greek         = GreekLetter, {Subscript};

Symbolic      = (Symbol | Shift), {(Symbol | Shift)}, {Subscript}
              | Punctuation, Subscript, {Subscript};
```

[To maximise the flexibility of the language, particularly when used for the metatheory of itself or of other languages even a punctuation character can be used to form a symbolic identifier by attaching a subscript.]

[Since the mandatory Greek characters are insufficient for actually typing real Greek words (there being no breathings etc.), the view is taken that Greek letters work as in ordinary mathematics, $\alpha\beta\gamma$ containing three names. This seems to be a good compromise, and works nicely with λ, μ, identifiers etc.]

Decoration Decoration comprises just a sequence of stroke characters:

```
Decoration = Stroke,{Stroke};
```

[We are assuming that proposal *Decor*.2 is adopted and that it is "implemented" in the transformation into abstract syntax. *Decor*.3 is equally simple, and essentially just says that decoration is allowed at the end of an identifier as part of the identifier.]

Numbers A numeric literal is a non-empty sequence of decimal digits:

```
Number = Digit, {Digit};
```

Strings A string literal denotes a sequence of arbitary text:

String = ?*ImplementationDependent*?;

Narrative The means for delimiting the narrative sections between formal material in a Z document is not defined in this standard:

Narrative = ?*ImplementationDependent*?;

Punctuation This kind of token includes the stop and box characters of section B.3 symbols.

```
Punctuation = Stop
            | Box;
```

Space A space token is a sequence of one or more white space characters.

Space = Format, {Format};

B.3 Character Set

At the most primitive level, a physical object (e.g, a document on paper or stored electronically) is interpreted as a finite sequence of *characters*. The method of deriving a sequence of characters from a physical object is not defined in this standard, however this section places minimum requirements on the character set.

The character set must include, at least, the characters in the sets *Letter*, *Greek*, *Digit*, *Symbol*, *Stop*, *Stroke*, *Subscript*, *Shift*, *Box*, *Quote*, *Ascii* and *Format* described in the following table. Additional characters may be used and are to be taken as elements of the set *Symbol*.

Letter	A	B	C	D	E	F	G	H	I	J
	K	L	M	N	O	P	Q	R	S	T
	U	V	W	X	Y	Z				
	a	b	c	d	e	f	g	h	i	j
	k	l	m	n	o	p	q	r	s	t
	u	v	w	x	y	z				
GreekLetter	α	β	γ	δ	ε	ζ	η	θ	ι	κ
	λ	μ	ν	ξ	π	ρ	σ	τ	υ	φ
	χ	ψ	ω							
			Γ	Δ				Θ		
	Λ		Ξ	Π		Σ		Υ	Φ	
		Ψ	Ω							
Digit	0	1	2	3	4	5	6	7	8	9
InFun	↦	+	−	∪	\	⁀	⊎	∗	∩	↾
	⨟	∘	⊕	#	⩤	⩥	◁	▷		
InRel	≠	∉	⊆	⊂	≤	≥	<	>		
InGen	⇸	⟶	⤔	↣	⤀	↠	⤖	↔		
Symbol	⋃	⋂	∅	⦇	⦈	⊢	⁀/	≠	.	~
	⟦	⟧	[	]	{	}	⟨	⟩	/	¬
	∧	∨	⇒	⇔	=	∈	∀	∃	•	
	×	≙	&	⊢		::=				
Stop	,	;	:	(	)	F	N	P	Z	
Underscore	_									
Stroke	′	?	!							
Subscript	Subscripted forms of any of the above characters.									
Shift	↗	↕								
Box	AX	SCH	GEN	END	IS	ST	BAR			
Quote	"									
Ascii	A member of the ISO character set with code in the range 32 to 126.									
Format	A format character such as space, tab, line-break or page-break.									

[↗ and ↥ are characters to shift in and out of superscription. Transitive closure, reflexive-transitive closure and relational inverse can be written as ↗ + ↥, ↗ ∗ ↥ and ↗ ~ ↥, each of which is an identifier.]

[*Letter* might also include other fonts, e.g. italic or bold. If so, there is a question as to whether the standard should insist that, e.g., '*A*' be treated the same as '**A**'?]

[The Greek letter omicron is not mandatory since it looks like an 'o' in some fonts.]

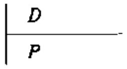
[The list of *Symbols* above should be extended in the actual standard to cover the requirements of the toolkit.]

[**AX**, **ST** etc. are intended to represent characters for drawing boxes of various sorts.]

B.4 Graphical Conventions

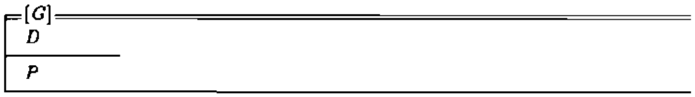
The following graphical conventions are adopted in this standard:

- The usual English orthographic conventions for interpreting printed text are assumed (division into pages and lines, direction of reding, ignoring page furniture such as headings and page numbers, identification of printed or written characters, and so on.)
- Sequences of non-Z text may be interspersed with Z text using any convention of presentation which allows the Z text to be unambiuonsly identified.
- Multiple newlines in succession are considered as one.
- A newline preceding or succeeding characters in the sets InFun, InRel, InGen and in Symbolis ignored.
- Characters in the set Subscript are written in the subscript position.
- The characters ↗ and ↥ delimit sequences of characters to be written in the superscript position.
- If G, D, P and S arbitrary sequences of characters not containing any of the box characters (**AX**, **BAR**, **ST**, **END**, **SCH** **GEN** and **IS**), then:
  - **AX** D **ST** P **END** is written as:



  - **AX** D **END** is written as:

  - **GEN** G **BAR** D **ST** P **END** is written as:



– GEN G BAR D END is written as:

```
┌═[G]════════════════════════════════════
│ D
└────────────────────────────────────────
```

– SCH S IS D ST P END is written as:

```
┌─S──────────────────────────────────────
│ D
├──────────
│ P
└────────────────────────────────────────
```

– and SCH S IS D END is written as:

```
┌─S──────────────────────────────────────
│ D
└────────────────────────────────────────
```

C Mathematical Toolkit

This section defines a Mathematical Toolkit or Library for nse with the Z notation. The principle is that those constructions that can be defined in terms of otbers are included in the Toolkit rather than in the core notation—this simplifies the core notation.

Most users will want to make use of the constructions defined in this section. This can therefore be regarded as a *basic* Toolkit, which nsers may augment with their own definitions, or replace if these definitions are not suitable for their use.

In this version of the Base Standard, the list of defined items follows the customary list of Toolkit items. Later versions of the Standard may include further definitions and explanations, and will link the Toolkit to related work on the semantics and proof system for Z.

Definitions of the Mathematical Toolkit are informally explained and illustrated. In some cases an illustration for one part of the Toolkit may rely on terms defined earlier in the toolkit. Many of the definitions given here are generic with respect to one or more sets.

Note: Instantiation of a generic definition can be performed with any appropriate sets, not necessarily the maximal sets of their types. However the informal descriptions of these definitions are often here expressed as if the sets used for instantiation were in fact types, since that is the way in which these definitions are commonly instantiated in Z specifications.

Reviewers of the draft standard are invited to comment on this approach.

C.1 Sets

Name

$\neq$ – Inequality

$\notin$ – Non-membership

Definition

$$
\begin{array}{|l}
[X] \\
\hline
\_ \neq \_ : X \leftrightarrow X \\
\_ \notin \_ : X \leftrightarrow \mathbb{P}\, X \\
\hline
\forall x, y : X \bullet x \neq y \Leftrightarrow \neg\, (x = y) \\
\forall x : X;\ S : \mathbb{P}\, X \bullet x \notin S \Leftrightarrow \neg\, (x \in S) \\
\hline
\end{array}
$$

Description

Inequality is a relation between values of the same type. The predicate $x \neq y$ denotes true when $x = y$ denotes false.

Non-membership is a relation between values of a certain type and sets of values of that type. The predicate $x \notin S$ denotes true when $x \in S$ denotes false.

Name

$\varnothing$ – Empty Set

$\subseteq$ – Subset relation

$\subset$ – Proper subset relation

$\mathbb{P}_1$ – Non-empty subsets

Definition

$\varnothing[X] == \{ x : X \mid \mathit{false} \}$

$$
\begin{array}{|l}
\hline
[X] \\
\_ \subseteq \_, \_ \subset \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X \\
\hline
\forall S, T : \mathbb{P}\, X \bullet \\
\quad (S \subseteq T \Leftrightarrow (\forall x : X \bullet x \in S \Rightarrow x \in T)) \land \\
\quad S \subset T \Leftrightarrow S \subseteq T \land S \neq T) \\
\hline
\end{array}
$$

$\mathbb{P}_1 X == \{ S : \mathbb{P}\, X \mid S \neq \varnothing \}$

Description

The empty set of values of a certain type is the set of values of that type that has no members.

If S and T are sets of values of the same type, then $S \subseteq T$ is a predicate denoting true if and only if every member of S is a member of T. The empty set of values of a certain type is a subset of every set of values of that type.

If S and T are sets of values of the same type, then $S \subset T$ is a predicate denoting true if and only if every member of S is a member of T and S and T are not equal. If S is a proper subset of T, then it is also a subset of T. The empty set of values of a certain type is a proper subset of every non-empty set of values of that type.

If X is a set, then $\mathbb{P}_1 X$ is the set of all non-empty subsets of X. $\mathbb{P}_1 X$ is a proper subset of $\mathbb{P}\, X$.

Name

$\cup$ – Set union

$\cap$ – Set intersection

$\setminus$ – Set difference

Definition

$$
\begin{array}{|l}
[X] \\
\hline
\_ \cup \_, \_ \cap \_, \_ \setminus \_ : \mathbb{P}\, X \times \mathbb{P}\, X \rightarrow \mathbb{P}\, X \\
\hline
\forall S, T : \mathbb{P}\, X \bullet \\
\quad S \cup T = \{\, x : X \mid x \in S \lor x \in T \,\} \land \\
\quad S \cap T = \{\, x : X \mid x \in S \land x \in T \,\} \lor \\
\quad S \setminus T = \{\, x : X \mid x \in S \land x \notin T \,\} \\
\hline
\end{array}
$$

Description

The union of two sets of values of the same type is the set of values that are members of either set.

The intersection of two sets of values of the same type is the set of values that are members of both sets.

The difference of two sets of values of the same types is the set of values that are members of the first set but not members of the second.

Name

$\bigcup$ – Generalized union

$\bigcap$ – Generalized intersection

Definition

$$
\begin{array}{|l}
[X] \\
\hline
\bigcup, \bigcap : \mathbb{P}(\mathbb{P}\, X) \rightarrow \mathbb{P}\, X \\
\hline
\forall A : \mathbb{P}(\mathbb{P}\, X) \bullet \\
\qquad \bigcup A = \{\, x : X \mid (\exists S : A \bullet x \in S) \,\} \land \\
\qquad \bigcap A = \{\, x : X \mid (\forall S : A \bullet x \in S) \,\} \\
\hline
\end{array}
$$

Description

The generalised union of a set of sets of values of the same type is the set of values of that type that are members of at least one of the sets.

The generalised intersection of a set of sets of values of the same type is the set of values of that type that are members of every one of the sets.

Name

first, *second* – Projection functions for ordered pairs

Definition

$$
\begin{array}{|l}
[X, Y] \\
\hline
\mathit{first} : X \times Y \rightarrow X \\
\mathit{second} : X \times Y \rightarrow Y \\
\hline
\forall x : X;\ y : Y \bullet \\
\qquad \mathit{first}(x, y) = x \land \\
\qquad \mathit{second}(x, y) = y \\
\hline
\end{array}
$$

Description

For any ordered pair (x, y), $\mathit{first}(x, y)$ is x and $\mathit{second}(x, y)$ is y.

If p is of type $X \times Y$, then $p = (\mathit{first}\ p, \mathit{second}\ p)$.

C.2 Relations

Name

$\leftrightarrow$ – Binary relations

$\mapsto$ – Maplet

Definition

$X \leftrightarrow Y == \mathbb{P}(X \times Y$

$$[X, Y]$$

$$\_ \mapsto \_ : X \times Y \rightarrow X \times Y$$

$$\begin{array}{l} \forall x : X;\ y : Y \bullet \\ \quad x \mapsto y = (x, y) \end{array}$$

Description

$X \leftrightarrow Y$ is the set of all sets of ordered pairs whose first members are members of X and whose second members are members of Y. To declare $R : X \leftrightarrow Y$ is to say that R is such a set of ordered pairs.

The maplet forms an ordered pair from two values, so if x is of type X and y is of type Y, then $x \mapsto y$ is of type $X \times Y$. $x \mapsto y$ is thus just another notation for (x, y).

Name

dom, ran – Domain and range of a relation

Definition

$$
\begin{array}{|l}
[X, Y] \\
\mathrm{dom} : (X \leftrightarrow Y) \rightarrow \mathbb{P}\, X \\
\mathrm{ran} : (X \leftrightarrow Y) \rightarrow \mathbb{P}\, Y \\
\hline
\forall R : X \leftrightarrow Y \bullet \\
\quad \mathrm{dom}\, R = \{\, x : X;\ y : Y \mid (x \mapsto y) \in R \bullet x \,\} \land \\
\quad \mathrm{ran}\, R = \{\, x : X;\ y : Y \mid (x \mapsto y) \in R \bullet y \,\}
\end{array}
$$

Description

The domain of a relation R is the set of first members of the ordered pairs in R. If R is of type $X \leftrightarrow Y$, the domain of R is of type $\mathbb{P}\, X$. If R is an empty relation, then its domain is an empty set.

The range of a relation R is the set of second members of the ordered pairs in R. If R is of type $X \leftrightarrow Y$, the domain of R is of type $\mathbb{P}\, Y$. If R is an empty relation, then its range is an empty set.

Name

id – Identity relation

; – Relational composition

∘ – Backward relational composition

Definition

$$\operatorname{id} X == \{ x : X \bullet x \mapsto x \}$$

$$\begin{array}{l}
[X, Y, X] \\
\hline
\_ ; \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \rightarrow (X \leftrightarrow Z) \\
\_ \circ \_ : (Y \leftrightarrow Z) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Z) \\
\hline
\forall R : X \leftrightarrow Y; S : Y \leftrightarrow Z \bullet \\
\quad R \,;\, S = S \circ R = \{ x : X; y : Y; z : Z \mid \\
\qquad\qquad (x \mapsto y) \in R \land (y \mapsto z) \in S \bullet x \mapsto z \}
\end{array}$$

Description

The identity relation on a set X is the relation that relates every member of X to itself. Its type is $X \leftrightarrow X$. The identity relation on an empty set is an empty relation.

The relational composition of a relation $R : X \leftrightarrow Y$ and $S : Y \leftrightarrow Z$ is a relation of type $X \leftrightarrow Z$ formed by taking all the pairs (x, y) of R whose second members are in the domain of S, and relating x to every member of Z that y is related to by S.

The backward composition of S and R is the same as the composition of R and S.

Name

$\lhd$ – Domain restriction

$\rhd$ – Range restriction

Definition

$$
\begin{array}{l}
[X, Y] \\
\_ \lhd \_ : \mathbb{P}\, X \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y) \\
\_ \rhd \_ : (X \leftrightarrow Y) \times \mathbb{P}\, Y \rightarrow (X \leftrightarrow Y) \\
\hline
\forall S : \mathbb{P}\, X;\ R : X \leftrightarrow Y \bullet \\
\quad S \lhd R = \{\ x : X;\ y : Y \mid x \in S \land (x \mapsto y) \in R \bullet x \mapsto y\ \} \\
\forall R : X \leftrightarrow Y;\ T : \mathbb{P}\, Y \bullet \\
\quad R \rhd T = \{\ x : X;\ y : Y \mid (x \mapsto y) \in R \land y \in T \bullet x \mapsto y\ \}
\end{array}
$$

Description

The domain restriction of a relation $R : X \leftrightarrow Y$ by a set $S : \mathbb{P}\, X$ is the set of pairs in R whose first members are in S. $S \lhd R$ is a subset of R, and its domain is a subset of S.

The range restriction of a relation $R : X \leftrightarrow Y$ by a set $T : \mathbb{P}\, Y$ is the set of pairs in R whose second members are in T. $R \rhd T$ is a subset of R, and its range is a subset of T.

Name

$⩤$ – Domain anti-restriction

$⩥$ – Range anti-restriction

Definition

$$
\begin{array}{l}
[X, Y] \\
\_ ⩤ \_ : \mathbb{P}\, X \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y) \\
\_ ⩥ \_ : (X \leftrightarrow Y) \times \mathbb{P}\, Y \rightarrow (X \leftrightarrow Y) \\
\hline
\forall S : \mathbb{P}\, X;\ R : X \leftrightarrow Y \bullet \\
\quad S ⩤ R = \{\, x : X;\ y : Y \mid x \notin S \wedge (x \mapsto y) \in R \bullet x \mapsto y \,\} \\
\forall R : X \leftrightarrow Y;\ T : \mathbb{P}\, Y \bullet \\
\quad R ⩥ T = \{\, x : X;\ y : Y \mid (x \mapsto y) \in R \wedge y \notin T \bullet x \mapsto y \,\}
\end{array}
$$

Description

The domain anti-restriction of a relation $R : X \leftrightarrow Y$ by a set $S : \mathbb{P}\, X$ is the set of pairs in R whose first members are not in S. $S ⩤ R$ is a subset of R, and its domain contains no members of S.

The range anti-restriction of a relation $R : X \leftrightarrow Y$ by a set $T : \mathbb{P}\, Y$ is the set of pairs in R whose second members are not in T. $R ⩥ T$ is a subset of R, and its range contains no members of T.

Name

$\_^{\sim}$ – relational inversion

Definition

$[X, Y]$

$\_^{\sim} : (X \leftrightarrow Y) \rightarrow (Y \leftrightarrow X)$

$\forall R : X \leftrightarrow Y \bullet$
$\quad R^{\sim} = \{ x : X ; y : Y \mid (x \mapsto y) \in R \bullet y \mapsto x \}$

Description

The inverse of a relation is the relation obtained by reversing every ordered pair in the relation.

Name

$\_(\!|\_|\!)$ – Relational image

Definition

$$
\begin{array}{|l}
[X, Y] \\
\hline
\_(\!|\_|\!) : (X \leftrightarrow Y) \times \mathbb{P}\, X \rightarrow \mathbb{P}\, Y \\
\hline
\forall R : X \leftrightarrow Y;\ S : \mathbb{P}\, X \bullet \\
\quad R(\!|S|\!) = \{\, x : X;\ y : Y \mid x \in S \land (x \mapsto y) \in R \bullet y \,\} \\
\hline
\end{array}
$$

Description

The relational image of a set $S : \mathbb{P}\, X$ under a relation $R : X \leftrightarrow Y$ is the set of values of type Y that are related under R to a value in S.

Name

$\_^+$ – Transitive closure

$\_^*$ – Reflexive-transitive closure

Definition

$[X]$

$\_^+, \_^* : (X \leftrightarrow X) \rightarrow (X \leftrightarrow X)$

$$\begin{array}{l} \forall R : X \leftrightarrow X \bullet \\ \quad R^+ = \bigcap\{\ Q : X \leftrightarrow X \mid R \subseteq Q \land Q \mathbin{;} Q \subseteq Q\ \} \land \\ \quad R^* = \bigcap\{\ Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \land R \subseteq Q \land Q \mathbin{;} Q \subseteq Q\ \} \end{array}$$

Description

The transitive closure of a relation $R : X \leftrightarrow X$ is the relation obtained by relating each member of the domain of R to its images under R, and to anything related to any of its images under R by any number of steps of application of R.

The reflexive transitive closure of a relation $R : X \leftrightarrow X$ is the relation formed by extending the transitive closure of R by the identity relation on X.

C.3 Functions

Name

$\pfun$ – Partial functions

$\fun$ – Total functions

Definition

$$
\begin{aligned}
&X \pfun Y == \\
&\quad \{\, f : X \rel Y \mid (\forall x : X;\ y_1, y_2 : Y \bullet \\
&\qquad\qquad\qquad (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2) \,\} \\
&X \fun Y == \{\, f : X \pfun Y \mid \operatorname{dom} f = X \,\}
\end{aligned}
$$

Description

The partial functions from X to Y are a subset of the relations $X \rel Y$. They are distinguished by the property that each x in X is related to at most one y in Y. $X \pfun Y$ is the set of all partial functions from X to Y, and to declare $f : X \pfun Y$ is to say that f is one such partial function.

The total functions from X to Y are a subset of the partial functions $X \pfun Y$. They are distinguished by the property that each x in X is related to exactly one y in Y. $X \fun Y$ is the set of all total functions from X to Y, and to declare $f : X \fun Y$ is to say that f is one such total function. The domain of $f : X \fun Y$ is X.

Name

$⤔$ – Partial injections

$\rightarrowtail$ – Total injections

Definition

$$X ⤔ Y ==$$
$$\{ f : X ⇸ Y \mid (\forall x_1, x_2 : \operatorname{dom} f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2) \}$$

$$X \rightarrowtail Y == (X ⤔ Y) \cap (X \rightarrow Y)$$

Description

The partial injections from X to Y are a subset of the partial functions $X ⇸ Y$. They are distinguished by the property that each y in Y is related to at most one x in X. Thus the inverse of a partial injection is also a partial injection. $X ⤔ Y$ is the set of all partial injections from X to Y, and to declare $f : X ⤔ Y$ is to say that f is one such partial injection.

The total injections from X to Y are a subset of the partial injections $X ⤔ Y$. They are distinguished by the property that each x in X is related to exactly one y in Y. $X \rightarrowtail Y$ is the set of all total injections from X to Y, and to declare $f : X \rightarrowtail Y$ is to say that f is one such total injection.

Name

$⤀$ – Partial surjections

$\twoheadrightarrow$ – Total surjections

$⤖$ – Bijections

Definition

$X ⤀ Y == \{ f : X ⇸ Y \mid \operatorname{ran} f = Y \}$
$X \twoheadrightarrow Y == (X ⤀ Y) \cap (X \rightarrow Y)$
$X ⤖ Y == (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$

Description

The partial surjections from X to Y are a subset of the partial functions $X ⇸ Y$. They are distinguished by the property that each y in Y is related to at least one x in X. $X ⤀ Y$ is the set of all partial surjections from X to Y, and to declare $f : X ⤀ Y$ is to say that f is one such partial surjection.

The total surjections from X to Y are a subset of the partial surjections $X ⤀ Y$. They are distiuguished by the property that each x in X is related to exactly one y in Y. $X \twoheadrightarrow Y$ is the set of all total surjections from X to Y, and to declare $f : X \twoheadrightarrow Y$ is to say that f is one such total surjection.

The bijections from X to Y are a subset of the total surjections $X \twoheadrightarrow Y$. They are distinguished by the property that each y in Y is related to exactly one x in X. $X ⤖ Y$ is the set of all bijections from X to Y, and to declare $f : X ⤖ Y$ is to say that f is one such total hijectiou.

C.3.1 Name

$\oplus$ – Functional overriding

C.3.2 Definition

$$
\begin{array}{|l}
[X, Y] \\
\_ \oplus \_ : (X \pfun Y) \times (X \pfun Y) \to (X \pfun Y) \\
\hline
\forall f, g : X \pfun Y \bullet \\
\qquad f \oplus g = ((\operatorname{dom} g) \ndres f) \cup g \\
\hline
\end{array}
$$

Description

If f and g are both functions from X to Y, then the functional overriding of f by g is the function g together with such pairs of f as have first elements different from the first element of any pair in g.

C.4 Numbers and finiteness

Name

$\mathbb{N}$	– Natural numbers
$\mathbb{Z}$	– Integers
$+, -, *, \mathrm{div}, \mathrm{mod}$	– Arithmetic operations
$<, \leq, \geq, >$	– Numerical comparison

Definition

$[\mathbb{Z}]$

$$\begin{array}{|l}
\mathbb{N} : \mathbb{P}\,\mathbb{Z} \\
\_+\_, \_-\_, \_*\_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
\_\,\mathrm{div}\,\_, \_\,\mathrm{mod}\,\_ : \mathbb{Z} \times (\mathbb{Z} \setminus \{0\}) \to \mathbb{Z} \\
-\_ : \mathbb{Z} \to \mathbb{Z} \\
\_<\_, \_\leq\_, \_\geq\_, \_>\_ : \mathbb{Z} \leftrightarrow \mathbb{Z} \\
\hline
\mathbb{N} = \{\, n : \mathbb{Z} \mid n \geq 0 \,\} \\
\text{... other definitions omitted...}
\end{array}$$

Description

The natural numbers are the integers from zero upwards. The type of $\mathbb{N}$ is $\mathbb{P}\,\mathbb{Z}$, since $\mathbb{N}$ is a set of integers. The declaration $n : \mathbb{N}$ makes $\mathbb{Z}$ the type of n, and entails the property $n \geq 0$.

Name

$\mathbb{N}_1$	– Strictly positive integers
$succ$	– Successor function

Definition

$\mathbb{N}_1 == \mathbb{N} \setminus \{0\}$

$$\begin{array}{|l} succ : \mathbb{N} \rightarrow \mathbb{N} \\ \hline \forall n : \mathbb{N} \bullet succ(n) = n + 1 \end{array}$$

Description

The strictly positive numbers $\mathbb{N}$ are the natural numbers except zero.

The successor of any natural number is the next natural number in ascending order.

Name

R^k – Iteration

Definition

$$
\begin{array}{|l}
\hline
[X] \\
iter : \mathbb{Z} \rightarrow (X \leftrightarrow X) \rightarrow (X \leftrightarrow X) \\
\hline
\forall R : X \leftrightarrow X \bullet \\
\qquad iter\ 0\ R = \mathrm{id}\, X \land \\
\qquad (\forall k : \mathbb{N} \bullet iter\ (k+1) R = R \mathbin{;} (iter\ k\ R)) \land \\
\qquad (\forall k : \mathbb{N} \bullet iter\ (-k) R = iter\ k\ (R^{\sim})) \\
\hline
\end{array}
$$

Description

The iteration of a relation $R : X \leftrightarrow X$ by zero is the identity relation on the set X. The iteration of a relation $R : X \leftrightarrow X$ by one is the relation R. The iteration of a relation $R : X \leftrightarrow X$ by an integer greater than one is the composition of R with its iteration by the next lower integer. The iteration of a relation $R : X \leftrightarrow X$ by an integer less that zero is the iteration of the inverse of R by the corresponding positive integer. Thus the iteration of R by -1 is the inverse of R.

The form: $iter\ k\ R$ is usually written R^k.

Name

$\,..\,$ – Number range

Definition

$$
\begin{array}{|l}
\_\,..\,\_ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{P}\,\mathbb{Z} \\
\hline
\forall a, b : \mathbb{Z} \bullet \\
\quad a\,..\,b = \{\, k : \mathbb{Z} \mid a \leq k \leq b \,\}
\end{array}
$$

Description

If a and b are integers, and a is less than b, the number range $a..b$ contains a, b and any integers between. If a is equal to b, the number range $a..b$ is a singleton set containing a only. If a is greater than b, the number range $a..b$ is an empty set of integers. The number range $a..b$ is always finite, and if $b \geq a$ its size is $b - a + 1$.

C MATHEMATICAL TOOLKIT

Name

$\mathbb{F}$	–	Finite sets
$\mathbb{F}_1$	–	Non-empty finite sets
$\#$	–	Number of members of a set

Definition

$$\mathbb{F}\,X == \{\, S : \mathbb{P}\,X \mid \exists\, n : \mathbb{N} \bullet \exists\, f : 1\,..\,n \rightarrowtail S \bullet \operatorname{ran} f = S \,\}$$
$$\mathbb{F}_1\,X == \mathbb{F}\,X \setminus \{\varnothing\}$$

$$[X]$$
$$\# : \mathbb{F}\,X \rightarrow \mathbb{N}$$
$$\forall\, S : \mathbb{F}\,X \bullet$$
$$\qquad \#S = (\mu\, n : \mathbb{N} \mid (\exists\, f : 1\,..\,n \rightarrowtail S \bullet \operatorname{ran} f = S))$$

Description

A set is finite if its members can be put into one-to-one correspondence with the natural numbers from 1 up to some limit. $\mathbb{F}\,X$ is the set of all finite subsets of X. $\mathbb{F}\,X$ is a subset of $\mathbb{P}\,X$. If X is finite, then it is a member of $\mathbb{F}\,X$.

The non-empty finite subsets of X are the finite subsets of X except the empty set.

The number of members of a finite set is the upper limit of the number range starting with 1 that can be put into one-to-one correspondence with the members of the set.

Name

$⇻$ – Finite partial functions

$⤕$ – Finite partial injections

Definition

$$X ⇻ Y == \{ f : X ⇸ Y \mid \mathrm{dom}\, f \in \mathbb{F}\, X \}$$
$$X ⤕ Y == (X ⇻ Y) \cap (X ⤔ Y)$$

Description

The finite partial functions from X to Y are the partial functions from X to Y whose domains are finite sets.

The finite partial injections from X to Y are the partial injections from X to Y whose domains are finite sets.

Name

min, *max* – Minimum and maximum of a set of numbers

Definition

$$
\begin{array}{|l}
min : \mathbb{P}_1 \mathbb{Z} \twoheadrightarrow \mathbb{Z} \\
max : \mathbb{P}_1 \mathbb{Z} \twoheadrightarrow \mathbb{Z} \\
\hline
min = \{ S : \mathbb{P}_1 \mathbb{Z};\ m : \mathbb{Z} \mid \\
\qquad m \in S \land (\forall n : S \bullet m \leq n) \bullet S \mapsto m \} \\
max = \{ S : \mathbb{P}_1 \mathbb{Z};\ m : \mathbb{Z} \mid \\
\qquad m \in S \land (\forall n : S \bullet m \geq n) \bullet S \mapsto m \}
\end{array}
$$

Description

The minimum of a non-empty set of integers that has a least member is the least member. Sets of integers that have no least member are not in the domain of *min*. If $a \leq b$, min $a..b = a$.

The maximum of a non-empty set of integers that has a greatest member is the greatest member. Sets of integers that have no greatest member are not in the domain of max. If $a \leq b$, max $a..b = b$.

C.5 Sequences

Name

seq	–	Finite sequences
seq_1	–	Non-empty finite sequences
iseq	–	Injective sequences

Definition

$$\begin{aligned}
\text{seq}\,X &== \{\, f : \mathbb{N} \nrightarrow\!\!\!\!\!\nrightarrow X \mid \text{dom}\, f = 1 \,..\, \#f \,\} \\
\text{seq}_1 &== \{\, f : \text{seq}\, X \mid \#f > 0 \,\} \\
\text{iseq}\, X &== \text{seq}\, X \cap (\mathbb{N} \rightarrowtail X)
\end{aligned}$$

Description

A sequence is a finite aggregate of values of the same type in which each value can be identified by its position in the sequence. The formal definition establishes a sequence as a partial function relating the numbers from the set 1..n for some n (the domain of the sequence) to the values (the range of the sequence). seq X is the set of all finite sequences of values of type X. The declaration S : seq X says that S is one such finite sequence. Since a sequence is a *function* (i.e. a set of ordered pairs), a sequence might be empty, and the function application notation $S\ i$ can be used to denote the element at position i, provided that i is in the domain of the sequence.

$\text{seq}_1\, X$ is the set of all non-empty finite sequences of values of type X. The declaration $s : \text{seq}_1\, X$ says that s is such a non-empty finite sequence. $\text{seq}_1\, X$ is a subset of seq X.

iseq X is the set of all injective finite sequences of values of type X. A sequence is injective if no value appears more than once in the sequence. The declaration S : iseq X says that S is such an injective finite sequence. iseq X is a subset of seq X.

Name

$\frown$ – Concatenation

Definition

$$
\begin{array}{|l}
[X] \\
\hline
\_ \frown \_ : \operatorname{seq} X \times \operatorname{seq} X \to \operatorname{seq} X \\
\hline
\forall s, t : \operatorname{seq} X \bullet \\
\quad s \frown t = s \cup \{ n : \operatorname{dom} t \bullet n + \# s \mapsto t(n) \} \\
\hline
\end{array}
$$

Description

Concatenation is a function of a pair of sequences of values of the same type that denotes a sequence that begins with the first sequence and continues with the second. Either or both of the sequences might be empty. If either sequence is empty, the result is the other sequence.

Name

head, *last*, *tail*, *front* – Sequence decomposition

Definition

$$
\begin{array}{l}
[X] \\
\hline
head, last : \mathrm{seq}_1\, X \rightarrow X \\
tail, front : \mathrm{seq}_1\, X \rightarrow \mathrm{seq}\, X \\
\hline
\forall s : \mathrm{seq}_1\, X \bullet \\
\quad head\ s = s(1) \land \\
\quad last\ s = s(\#s) \land \\
\quad tail\ s = (\lambda\, n : 1\,..\,\#s - 1 \bullet s(n+1)) \land \\
\quad front\ s = (1\,..\,\#s - 1) \lhd s
\end{array}
$$

Description

If S is a non-empty sequence of values of type X, then *head* S is the value of type X that is first in the sequence. Empty sequences are not in the domain of *head*.

If S is a non-empty sequence of values of type X, then *last* S is the value of type X that is last in the sequence. Empty seqnences are not in the domain of *last*.

If S is a non-empty sequence of values of type X, then *tail* S is the sequence of values of type X obtained from S by discarding the first member. Empty sequences are not in the domain of *tail*.

If S is a non-empty sequence of values of type X, then *front* S is the sequence of values of type X obtained from S by discarding the last member. Empty sequences are not in the domain of *front*.

Name

rev — reverse

Definition

$$
\begin{array}{l}
[X] \\
\hline
rev : \mathrm{seq}\, X \to \mathrm{seq}\, X \\
\hline
\forall s : \mathrm{seq}\, X \bullet \\
\quad rev\ s = (\lambda n : \mathrm{dom}\, s \bullet s(\#s - n + 1))
\end{array}
$$

Description

The reverse of a sequence is the sequence obtained by taking its members in the opposite order.

Name

$\restriction$ – Filtering

Definition

$$
\begin{array}{|l}
[X] \\
\hline
\_ \restriction \_ : \mathrm{seq}\, X \times \mathbb{P}\, X \to \mathrm{seq}\, X \\
\hline
\forall V : \mathbb{P}\, X \bullet \\
\quad \langle\rangle \restriction V = \langle\rangle \land \\
\quad (\forall x : X \bullet \\
\qquad (x \in V \Rightarrow \langle x\rangle \restriction V = \langle x\rangle) \land \\
\qquad (x \notin V \Rightarrow \langle x\rangle \restriction V = \langle\rangle)) \land \\
\quad (\forall s, t : \mathrm{seq}\, X \bullet \\
\qquad ((s \frown t) \restriction V = (s \restriction V) \frown (t \restriction V))
\end{array}
$$

Description

The filter of a sequence of values of type X by a set of values of type X is the sequence obtained from the original by discarding any members that are not in the set.

Name

$\frown/$ – Distributed concatenation

Definition

$$
\begin{array}{|l}
[X] \\
\frown/ : \mathrm{seq}(\mathrm{seq}\, X) \to \mathrm{seq}\, X \\
\hline
\frown/\langle\rangle = \langle\rangle \\
\forall s : \mathrm{seq}\, X \bullet \frown/\langle s\rangle = s \\
\forall q, r : \mathrm{seq}(\mathrm{seq}\, X) \bullet \\
\quad \frown/(q \frown r) = (\frown/\, q) \frown (\frown/\, r)
\end{array}
$$

Description

The distributed concatenation of a sequence of sequences of values of type X is a sequence of values of type X obtained by concatenating the lesser sequences in the order in which they appear in the greater sequence.

Name

disjoint – Disjointness

partition – Partitions

Definition

$$
\begin{array}{l}
[I, X] \\
\hline
\mathrm{disjoint}\ \_ : \mathbb{P}(I \nrightarrow \mathbb{P}\, X) \\
\_\ \mathrm{partition}\ \_ : (I \nrightarrow \mathbb{P}\, X) \leftrightarrow \mathbb{P}\, X \\
\hline
\forall S : I \nrightarrow \mathbb{P}\, X;\ T : \mathbb{P}\, X \bullet \\
\quad (\mathrm{disjoint}\ S \Leftrightarrow \\
\quad\quad (\forall i, j : \mathrm{dom}\, S \mid i \neq j \bullet S(i) \cap S(j) = \varnothing)) \land \\
\quad (S\ \mathrm{partition}\ T \Leftrightarrow \\
\quad\quad \mathrm{disjoint}\ S \land \bigcup\{\, i : \mathrm{dom}\, S \bullet S(i)\,\} = T)
\end{array}
$$

Description

An indexed family of sets is disjoint if no two members having distinct indexes have any members in common.

An indexed family S of sets partition a set T if S is disjoint and the union of all the members of S is T.

C.6 Bags

Name

bag	–	Bags
count	–	Multiplicity
in	–	Bag membership

Definition

$\mathrm{bag}\, X == X \pfun \mathbb{N}_1$

$$
\begin{array}{|l}
[X] \\
\hline
count : \mathrm{bag}\, X \rightarrowtail (X \rightarrow \mathbb{N}) \\
\_\ \mathrm{in}\ \_ : X \leftrightarrow \mathrm{bag}\, X \\
\hline
\forall x : X;\ B : \mathrm{bag}\, X \bullet \\
\qquad count\ B = (\lambda\, x : X \bullet 0) \oplus B \land \\
\qquad x\ \mathrm{in}\ B \Leftrightarrow x \in \mathrm{dom}\, B \\
\hline
\end{array}
$$

Description

A bag represents an aggregate in which order is not important, but in which a given value can occur several times. A bag of values of type X is a function whose domain is a subset of X and whose range is a set of strictly positive natural numbers.

The count of a bag of values of type X is a function that extends the bag function by relating every member of X that is not is the domain of the bag to zero.

A value $x : X$ is said to be in $B : bagX$ if and only if x is in the domain of B.

Name

$\uplus$ – Bag union

Definition

$$\begin{array}{|l}
[X] \\
\hline
\_ \uplus \_ : \mathrm{bag}\, X \times \mathrm{bag}\, X \rightarrow \mathrm{bag}\, X \\
\hline
\forall B, C : \mathrm{bag}\, X;\ x : X \bullet \\
\qquad \mathit{count}\ (B \uplus C) x = \mathit{count}\ B\ x + \mathit{count}\ C\ x \\
\hline
\end{array}$$

Description

The bag union of two bags is the bag that relates every member of the domain of either bag to the sum of its occurrences in the two bags.

Name

items – Bag of elements of a sequence

Definition

$$
\begin{array}{l}
[X] \\
\hline
items : \mathrm{seq}\, X \rightarrow \mathrm{bag}\, X \\
\hline
\forall s : \mathrm{seq}\, X;\ x : X \bullet \\
\quad count\ (items\ s)x = \#\{\ i : \mathrm{dom}\ s \mid s(i) = x\ \}
\end{array}
$$

Description

The items of a sequence of values of type X is a bag such that the range of the sequence and the domain of the bag are the same, and the each value in the domain of the bag is related to the number of indexes in the sequence at which that value occurred.

D Z Interchange Format

D.1 Introduction

The **Z Interchange Format** defines a portable representation of Z, allowing Z documents to be transmitted between different machines. The most suitable means of communication between different machines is by using text files in which the character set is limited for portability reasons. The Interchange Format defines a syntax for such text files.

The basis for the Interchange Format is the ISO Standard Generalized Markup Language (SGML). SGML permits the structure of texts to be represented and encoded in a standard form, convenient for storage, editing, retrieval and processing. The SGML Standard is defined in [11]. A general description of the aims and principles of SGML, together with an annotated version of the standard, is included in *The SGML Handbook* by C. F. Goldfarb [8]. Case studies and applications in SGML are described in the work of the Text Encoding Initiative as reported in [24].

The structure of this Appendix is as follows:

- the first section describes the scope of the Interchange Format — i.e. the facilities offered by the Format.
- the second section contains an informal description of SGML.
- the next section defines the Interchange Format.
- the final section presents explanatory material and examples of the use of the Interchange Format.

D.2 Scope of the Interchange Format

The Interchange Format allows a distinction to be made between formal text and other text included in a Z document. The Interchange Format does not prescribe the structure of all parts of a Z document, and does not define the internal structure of informal text.

As one possible application of the Z Interchange Format is to send a Z document to another machine for Z syntax checking, the format is sufficiently liberal to permit syntactically incorrect Z to be written. The format thus prescribes markup only for the higher levels of the Z syntax hierarchy; in most cases this is at the level of a Z paragraph, although for axiomatic and 'boxed' definitions there is scope for creating a more detailed markup if desired.

For a Z document to be syntactically correct when written in the Interchange Format, it must conform at the higher levels to the markup defined in this Appendix, and at the lower levels (e.g. predicate or expression level) to the Z Concrete Syntax, with all mathematical symbols replaced by the alphanumeric representations defined in Section D.4.3.

The Interchange Format also provides markup for requirements which are additional to the prime requirement for encoding the *structure* of the Z in a document. The following requirements are accommodated:

- identification of informal Z fragments, i.e. Z fragments which do not contribute to the *formal* part of a Z document;

- definition of the fixity and binding priority (where applicable) of user-defined names;
- allocation of unique identifiers to Z paragraphs, e.g. so that associations between Z operation schemas and data-flow diagrams can be made, or so that Z definitions can be indexed;
- logical grouping of Z paragraphs independently of the positions they occupy in the document, e.g. so that the group can be considered as a unit for type-checking purposes, or that 'units of conservative extension' can be identified for subsequent processing by a proof tool;
- labelling of 'stacked' predicates in an axiomatic or 'boxed' definition.

D.3 Introduction to SGML

This section provides an introduction to SGML, sufficient for the understanding of the definition of the Interchange Format in Section D.4. More comprehensive descriptions of SGML are given in [11] and [8].

Examples of text written in SGML are printed with a fixed-width font (the **tt** font in LaTeX) as follows:

```
<tag> text </tag>
```

D.3.1 SGML Element Definitions

Structures are described in the Interchange Format by means of SGML **elements**. Elements are delimited by start-tags and end-tags. A start-tag is of the form <name>, where name is the generic identifier of the delimited element. The end-tag is of the form </name>. For example, a particular Z given set definition may be written in the Interchange Format as:

```
<givendef> NAME, DATE </givendef>
```

The internal structure of a general SGML element is itself defined in SGML by means of a formal SGML **element declaration**. The components of an element declaration are:

1. the name of the element;
2. two characters (separated by a space) which specify the **minimisation rules** for the element;
3. the **content model** of the element.

The minimisation rules indicate whether the start-tags or end-tags may be omitted in instances of the element. The first character in the pair corresponds to the start-tag and the second to the end-tag. The character '-' or 'o' indicates that the corresponding tag respectively must be present or may be omitted.

The content model specifies what occurrences of the element may legitimately contain. Contents may be specified in terms of other elements and of special reserved words. Ultimately all elements consist of 'parsed character data' (represented in element declarations by the reserved word #PCDATA), which contains any valid character data but *not* further elements. Further structural information concerning elements which are constituents of the declared element is provided by the use of **occurrence indicators** and **group connectors**.

Occurrence indicators define how many times a constituent element may occur in instances of the defined element and are placed at the end of the constituent element. The following occurrence indicators are used in this Appendix:

- a question mark (?) indicates that the preceding element occurs at most once;
- an asterisk (*) indicates that the preceding element may be absent or occurs one or more times;
- a plus sign (+) indicates that the preceding element occurs one or more times.

Group connectors specify the ordering of constituent elements. The following connectors are nsed in this Appendix:

- a vertical bar (|) indicates that only one of the components it connects may appear;
- a comma (,) indicates that the components mnst appear in that order.

For example the element definition for a Z schema declaration is given as:

```
<!ELEMENT schemadef    - -
(#PCDATA, sub?, formals?,
(sexp | (decpart, axpart?))) >
```

Occurrences of this element thus consist of parsed character data (representing the name of the schema), followed by an optional subscript, followed by an optional element which holds the formal parameters of the definition, followed by either an element representing a schema expression or a constrnct representing the declaration part and (optional) axiomatic part of a schema definition. The start-tag and end-tag of the schema definition must both be present.

D.3.2 SGML attribute declarations

In SGML, **attributes** are nsed to provide information associated with elements. The Interchange Format employs attributes to encode layont information and other information which is not considered to be part of the *structure* of a Z specification. For example, the Interchange Format defines a 'style' attribute for schema definitions which permits an indication of whether the definition should be in vertical or horizontal form. An occnrrence of a 'schemadef' element may thus contain an **attribute-value pair** inside the element's start-tag; for example:

```
<schemadef style=vert> S ...  </schemadef>
```

An SGML **attribute declaration** specifies the name(s) of the element(s) to which the attributes are attached, followed by a list of rows, each of which consists of the name of the attribnte being declared, its type, and an optional default value. A type may be given as a collection of explicit values, or as one of the following special keywords:

`CDATA`	the attribute valne may contain any valid character data and must be delimited by double quotation marks;

`ID`	indicates that a unique identifying value will be snpplied for each instance of the element;
`NMTOKEN`	the attribute value is a name token (i.e. any alphanumeric string).

The default value for an attribute may be denoted as one of the set of explicit values defined for an attrihute; alternatively it may be one of the following special values:

`#IMPLIED`	a value need not be supplied;
`#REQUIRED`	a value mnst be supplied.

D.3.3 SGML entities

An **SGML entity** is a named part of a marked-up document. An example of an entity declaration is:

```
        <!ENTITY  ZBS      ''Z Base Standard, version 1.0'' >
```

References to entities are contructed by prefixing the name of the entity with an ampersand character (&) and delimiting the end of the name with a semicolon, space or end-of-file. Here is an example of an entity reference:

```
        We are now in a position to issue the &ZBS;.
```

The entity reference in this document fragment would be expanded by an SGML parser as:

We are now in a position to issue the Z Base Standard, version 1.0.

In the Interchange Format, SGML entities are used to represent non-alphanumeric Z symbols. When an SGML parser is nsed to analyse a Z document, association between the alphanumeric representation of mathematical symhols and their local code are recorded in SGML entity declarations. Since local word processor codes may differ for different Z users, Section D.4.3 records the entity names used in the Interchange Format, together with the normal representations of corresponding Z symbols.

D.4 Definition of the Interchange Format

This section presents the definition of the Interchange Format as a collection of SGML declarations. Explanatory material and examples of the use of the Interchange Format are given below.

An SGML Document Type Definition (DTD) defines the syntax of SGML-conformant documents in a style which is readable by SGML parsers. The Interchange Format does not warrant a full DTD for two reasons:

- the format does not specify the structure of the informal text in a Z document;
- the entity declarations are implementation-dependent.

A DTD consists of a header, followed by a body containing the element declarations, attribute declarations and entity declarations. The definition of the Interchange Format presented in this Section may be considered as the partial body of a DTD (*partial* because the entity declarations are not given explicitly); it is also equivalent to a definition in BNF of the structure of the Interchange Format. Newlines are not significant in the Interchange Format except when they serve to separate predicates or declarations.

Incidentally, it is unlikely that the interchange format could ever accommodate every function required by its users. In the SGML scheme, any collection of SGML declarations (such as those which define this Interchange Format) may be replaced or enhanced by the pre-insertion of additional SGML declarations. Such a 'customisation' of the Interchange Format would be acceptable by SGML parsers.

D.4.1 Element declarations

These declarations define the higher-level structure of the Z paragraphs in a Z document written in the Interchange Format. It corresponds closely to the Z Concrete Syntax, apart from the introduction of two high-level structures (*i.e.* **opdec** and **infundec**) which are used by the author of a Z document to define any special fixity and priority of symbols and names declared in the document.

Note that it is possible to identify the individual 'stacked' predicates (*i.e.* a collection of predicates separated by newlines) in the predicate part of a boxed definition. This facility is optional; the complete stack of predicates may be identified as a single predicate if that is more convenient (e.g. if the originator of the document has no automatic translator to the Interchange Format which recognises significant newlines).

Element definitions are provided for the representation of superscripts and subscripts.

```
<!ELEMENT Z      - -
(opdec | infundec | givendef | axdef | constraint
| schemadef | gendef | abbrevdef
| conjecture | structsetdef)* >

<!ELEMENT (informalZ | conjecture | constraint
| infundec | sup)     - -
(#PCDATA | string | sub | sup)+ >
```

```
<!ELEMENT (sexp | decpart | body | predicate)
- o    (#PCDATA | string | sub | sup)+ >

<!ELEMENT (givendef | infundec | opdec | formals | label)
- -    (#PCDATA | sub | sup)+ >

<!ELEMENT axdef    - -    (decpart, axpart?) >

<!ELEMENT schemadef    - -
(#PCDATA, sub?, formals?,
(sexp | (decpart, axpart?))) >

<!ELEMENT gendef    - -    (formals?, decpart, axpart?) >

<!ELEMENT (structsetdef | abbrevdef)    - -
((#PCDATA | sup | sub)+, body) >

<!ELEMENT axpart    - o    (predicate+) >

<!ELEMENT (string, sub)    - -    (#PCDATA) >
```

D.4.2 Attribute declarations

The attribute declarations permit the association of additional information with occurrences of elements in a Z document written in the Interchange Format.

The attributes **id** and **group** permit respectively unique identification and logical grouping of Z paragraphs.

The attributes **style** and **purpose** define respectively the layout and intended use of a schema definition.

The attribute **label** permits informal annotation of each member of the 'stack' of predicates which constitutes the axiomatic part of a boxed definition.

The attribute **optype** for the declaration of an operator symbol permits the association of a fixity with that symbol. This fixity applies to *all* occurrences of that symbol in the Z document.

NOTE TO EDITORS: This may not be the case in Version 0.6 of the Base Standard.

The attribute **priority** for the declaration of an infix function symbol permits the association of a binding priority with that symbol. This priority applies to *all* occurrences of that symbol in the Z document.

```
<!ATTLIST
(givendef | axdef | constraint | schemadef | gendef
| abbrevdef | structsetdef)
id    ID    #IMPLIED
group    NMTOKEN    #IMPLIED >

<!ATTLIST  schemadef
```

```
style    (vert | horiz)    horiz
purpose    (state | operation | datatype)    #IMPLIED >

<!ATTLIST  predicate     label    CDATA    #IMPLIED >

<!ATTLIST  opdec
optype    (ingen | inrel | pregen | prerel | postfun)
#REQUIRED >

<!ATTLIST  infundec
priority    (1 | 2 | 3 | 4 | 5 | 6)    6 >
```

D.4.3 Entity declarations

The entity declarations for the Interchange Format are not presented in the conventional SGML format because of the dependence of the internal representation of mathematical symbols on the implementation of each user's Z document processor. The mode of declaration used here is to present a table which records the association of each entity name with the corresponding mathematical symbol.

Many of the entity names defined here have already been defined as standard in Appendix D of [11]. Entity names which have been devised specifically for the Interchange Format are identified by an asterisk.

We first present the symbols of the basic Z language. The set of symbols covered by these definitions consists of those basic language symbols which are not subsumed by the Element Declarations presented in Section D.4.1. Entity names are not provided for the underscore (_), prime ('), colon (:), comma (,), query (?), shriek (!), period (.), unary minus (-), parenthesis, schema renaming (/) and equality (=) symbols, as it is assumed that these symbols, though non-alphanumeric, are reasonably portable.

INFORMAL NAME	ENTITY NAME	SYMBOL
left square bracket	lsqb	[
right square bracket	rsqb	]
left chevron bracket	lchev (*)	⟪
right chevron bracket	rchev (*)	⟫
bar	verbar	|
fat dot	bull	•
universal quantifier	forall	∀
existential quantifier	exist	∃
unique existential quantifier	exist1 (*)	$\exists_1$
membership	isin	∈
negation	not	¬
conjunction	and	∧
disjunction	or	∨
implication	rArr	⇒
equivalence	iff	⇔
power set	pset (*)	ℙ
theta	thetas	θ

Cartesian product	prod (*)	$\times$
mu	mu	μ
left set bracket	lcub	$\{$
right set bracket	rcub	$\}$
left sequence bracket	lseq (*)	$\langle$
right sequence bracket	rseq (*)	$\rangle$
left bag bracket	lbag (*)	$\llbracket$
right bag bracket	rbag (*)	$\rrbracket$
lambda	lambda	λ
left relational image bracket	limg (*)	⦇
right relational image bracket	rimg (*)	⦈
Delta	Delta	Δ
Xi	Xi	Ξ
alpha	alpha	α
beta	beta	β
gamma	gamma	γ
delta	delta	δ
epsilon	epsi	ϵ
zeta	zeta	ζ
eta	eta	η
iota	iota	ι
kappa	kappa	κ
nu	nu	ν
xi	xi	ξ
pi	pi	π
rho	rho	ρ
sigma	sigma	σ
tau	tau	τ
upsilon	upsi	υ
phi	phis	ϕ
chi	chi	χ
psi	psi	ψ
omega	omega	ω
Gamma	Gamma	Γ
Theta	Theta	Θ
Lambda	Lambda	Λ
Pi	Pi	Π
Sigma	Sigma	Σ
Upsilon	Upsi	Υ
Phi	Phi	Φ
Psi	Psi	Ψ
Omega	Omega	Ω
schema composition	scomp (*)	$;$
schema hiding	hide (*)	$\setminus$
schema projection	proj (*)	$\upharpoonright$
turnstile	turn (*)	$\vdash$
ampersand	amp	$\&$
binding	rarrw	$\leadsto$

left binding bracket	lbind (*)	⦉
right binding bracket	rbind (*)	⦊

D Z INTERCHANGE FORMAT

We now present the symbols of the Z mathematical toolkit. The symbols covered by these definitions are the non-alphanumeric members of the Z Mathematical Toolkit. Entity names are not provided for the addition (+) and multiplication (*) symbols, as it is assumed that these symbols, though non-alphanumeric, are reasonably portable.

NAME	ENTITY NAME	SYMBOL
inequality	ne	$\neq$
non-membership	notin	$\notin$
empty set	empty	$\varnothing, \{\ \}$
proper subset	sub	$\subset$
non-empty subsets	pset1 (*)	$\mathbb{P}_1$
subset	sube	$\subseteq$
set union	cup	$\cup$
set intersection	cap	$\cap$
set difference	sdiff (*)	$\setminus$
generalised union	Cup (*)	$\bigcup$
generalised intersection	Cap (*)	$\bigcap$
binary relation	rel (*)	$\leftrightarrow$
maplet	map (*)	$\mapsto$
(backward) composition	comp (*)	$;$
forward composition	compfn	$\circ$
domain restriction	dres (*)	$\lhd$
range restriction	rres (*)	$\rhd$
domain subtraction	dsub (*)	$⩤$
range subtraction	rsub (*)	$⩥$
relational inverse	tilde	$\sim$
transitive closure	tcl (*)	$+$
reflexive-transitive closure	rtcl (*)	$*$
partial functions	pfun (*)	$⇸$
total functions	tfun (*)	$\rightarrow$
partial injections	pinj (*)	$⤔$
total injections	tinj (*)	$\rightarrowtail$
partial surjections	psur (*)	$⤀$
total surjections	tsur (*)	$\twoheadrightarrow$
bijections	bij (*)	$⤖$
functional override	oplus	$\oplus$
natural numbers	Nat (*)	$\mathbb{N}$
integers	Int (*)	$\mathbb{Z}$
less than	lt	$<$
less than or equal to	le	$\leq$
greater than or equal to	ge	$\geq$
greater than	gt	$>$
strictly positive integers	Nat1 (*)	$\mathbb{N}_1$
number range	upto (*)	$..$
unary minus	uminus (*)	$-$
binary minus	bminus (*)	$-$
finite sets	fset (*)	$\mathbb{F}$

non-empty finite sets	fset1 (*)	$\mathbb{F}_1$
cardinality	num	#
finite partial functions	fpfun (*)	⇻
finite partial injections	fpinj (*)	⤕
filter	filter (*)	↾
concatenation	cat (*)	⁀
distributed concatenation	dcat (*)	⁀/
non-empty finite sequences	seq1 (*)	seq_1

NOTE TO EDITORS: These library members are taken from the 1st edition of the ZRM. We must add any new members.

D.5 Examples

This section presents examples of the use of the Interchange Format. Thses examples are carefully chosen to cover the more difficult aspects of the Format. The areas covered are indicated in the subsection headings.

D.5.1 Declaring Infix Identifiers

Consider the following axiomatic definition, which declares a relation *isTwice* which is intended to be used in an infix manner:

$$\_isTwice\_ : \mathbb{N} \leftrightarrow \mathbb{N}$$
$$\forall i, j : \mathbb{N} \bullet i \; isTwice \; j \Leftrightarrow i = 2 * j$$

The encoding of this Z definition in the Interchange Format includes not only the encoding of the axiomatic definition itself, but also an 'opdec' statement which declares the fixity of *isTwice*:

```
<Z>
<opdec optype=inrel> isTwice </opdec>

<axdef>
<decpart>
_isTwice_: &Nat &rel &Nat
<axpart>
<predicate>
&forall i, j: &Nat &bull i isTwice j &iff i = 2*j
</axdef>
</Z>
```

D.5.2 Subscripts and superscripts

The axiomatic definition

$$a_1, a_3 : \mathbb{N}$$
$$a_3 \; isTwice \; a_1^2$$

is encoded in the Interchange Format as:

```
<Z> <axdef>
<decpart>
a<sub> 1 </sub>, a<sub> 3 </sub>: &Nat
<axpart>
<predicate>
a<sub> 1 </sub> isTwice a<sub> 3 </sub><sup> 2 </sup>
</axdef> </Z>
```

D.5.3 Schema definitions and predicate labelling

Consider the following definitions:

$[PERSON, HOUSE]$

$$
\begin{array}{|l}
\hline
\textit{Street} \\
inhabits : PERSON \twoheadrightarrow HOUSE \\
houses : \mathbb{P}\, HOUSE \\
\hline
houses = \operatorname{ran} inhabits \\
\forall h : houses \bullet \# inhabits^{\sim}(\!|\{h\}|\!) \leq 4 \\
\quad \textit{/* No house may be occupied by more than 4 persons */} \\
\hline
\end{array}
$$

The author of this specification intends to accomplish the following objectives:

- to attach a label to the second predicate in the schema definition;
- to indicate that the schema definition should be displayed in vertical form;
- to indicate (to a specification checker, for example) that the schema *Street* defines the *state* of a system.

These objectives can be attained in the Interchange Format with the following encoding:

```
<Z>
<givendef> PERSON, HOUSE </givendef>

<schemadef style=vert purpose=state> Street
<decpart>
inhabits: PERSON &fpfun HOUSE
houses: &pset HOUSE
<axpart>
<predicate>
houses = &ran inhabits
<predicate
label=''No house may be occupied by more than 4 persons''>
&forall h: houses &bull
&num inhabits&inv&limg&lcub h&rcub&rimg &le 4
</schemadef>
</Z>
```

D.5.4 Abbreviation definitions

Note that in the Interchange Format there are no *entity* representations of the symbols immediately associated with top-level definitions such as structural set definitions and abbreviation definitions. These symbols are subsumed by the element tags for those definitions. For example, consider the following abbreviation definition:

$$n == 5 + x$$

This definition is encoded in the Interchange Format as:

```
<Z> <abbrevdef> n <body> 5 + x </abbrevdef> </Z>
```

E Z Character Set

NAME	SYNTAX TERMINALS
Given set brackets	[ ]
Schema definition	$\widehat{=}$
Abbreviation definition	$==$
Chevron Brackets	$\langle\!\langle \quad \rangle\!\rangle$
Bar	$\mid$
Schema brackets	[ ]
Colon	$:$
Semicolon	$;$
Comma	$,$
Fat dot	$\bullet$
Universal quantifier	$\forall$
Existential quantifier	$\exists$
Unique Existential quantifier	$\exists_1$
Equality	$=$
Membership	$\in$
Negation	$\neg$
Conjunction	$\land$
Disjunction	$\lor$
Implication	$\Rightarrow$
Equivalence	$\Leftrightarrow$
Power set	$\mathbb{P}$
Selection	$.$
Theta	θ
Cartesian product	$\times$
Tuple Brackets	$(\quad)$
Mu	μ
Set brackets	$\{ \quad \}$
Sequence brackets	$\langle \quad \rangle$
Bag brackets	$[\![\quad]\!]$
Lambda	λ
Relational image Brackets	$(\!| \quad |\!)$
Dash	$'$
Query	$?$
Shriek	$!$
Delta	Δ
Xi	Ξ

E Z CHARACTER SET

NAME	TOOLKIT SYMBOLS
Inequality	$\neq$
Non-membership	$\notin$
Empty-set	$\varnothing$, $\{\ \}$
Proper subset	$\subset$
Non-empty subsets	$\mathbb{P}_1$
Subset	$\subseteq$
Set union	$\cup$
Set intersection	$\cap$
Set difference	$\setminus$
Generalised union	$\bigcup$
Generalised intersection	$\bigcap$
Binary relation	↔
Maplet	$\mapsto$
Composition	⨾, $\circ$
Domain restriction	$\lhd$
Range restriction	$\rhd$
Domain subtraction	⩤
Range subtraction	⩥
Relational inverse	$R^{\sim}$
Transitive closure	R^{+}
Reflexive-transitive closure	R^{*}
Partial functions	⇸
Total functions	$\rightarrow$
Partial injections	⤔
Total injections	$\rightarrowtail$
Partial surjections	⤀
Total surjections	$\twoheadrightarrow$
Bijections	⤖
Functional override	$\oplus$
Natural numbers	$\mathbb{N}$
Integers	$\mathbb{Z}$
Addition	$+$
Subtraction	$-$
Mulitplication	$*$
Division	div
Less than	$<$
Less than or equal to	$\leq$
Greater than or equal to	$\geq$
Greater than	$>$
Strictly positive integers	$\mathbb{N}_1$
Relational iteration	R^k
Number range	$..$
Finite sets	$\mathbb{F}$
Non-empty finite sets	$\mathbb{F}_1$
Cardinality	#
Finite partial functions	⇻

Finite partial injections	⤕
Filter	↾
Concatenation	⁀

F A deductive system for Z

F.1 Introduction

This section presents a deductive system for Z. It is one of several possible deductive systems for Z, and has been developed as part of the ZIP project. There are two aspects to the choice of a deductive system: form and content. The form concerns the syntax and manner of conducting proofs. The content concerns the set of theorems that are deducible within the system.

The deductive system is a Gentzen-style sequent calculus in which sequents are composed of declarations and predicates. The rules of the logic are presented in a simplified form. The meta-theorems of the logic (theorems about the rules) permit the extension of the rules into a more practical form.

The loose definition of function application and definite description in the semantics permits a number of interpretations of their meanings. This deductive system is sound with respect to a model in which all well-typed expressions have a value.

F.2 Sequents

The basic building block for a sequent calculus is a sequent. A sequent is composed of an antecedent and a consequent.

$$\mathsf{Sequent} \;=\; \mathsf{Antecedent} \vdash \mathsf{Consequent}$$

The antecedent is a list of declarations separated by the symbol † and a list of predicates separated by commas.

$$\mathsf{Antecedent} \;=\; \mathsf{Declaration} \dagger \ldots \dagger \mathsf{Declaration} \mid \mathsf{Predicate}, \ldots, \mathsf{Predicate}$$

The consequent is also a list of predicates. The syntax for a consequent is the following:

$$\mathsf{Consequent} \;=\; \mathsf{Predicate}, \ldots, \mathsf{Predicate}$$

Thus a sequent appears as:

$$D_1 \dagger \ldots \dagger D_n \mid \Psi \vdash \Phi$$

where the meta variables Ψ and Φ represent lists of predicates. The lists of predicates in the antecedent and consequent are sets so the ordering is of no consequence.

A sequent is well-typed if the predicates are all well-typed in the environment enriched by the declarations where the declarations introduce new scope.

$$\llbracket D_1 \dagger \ldots \dagger D_m, \mid P_1, \ldots, P_n \vdash Q_1, \ldots, Q_l \rrbracket^{T} \;=\; \operatorname{dom}\left(\llbracket D_1 \rrbracket^{T} ; \ldots ; \llbracket D_m \rrbracket^{T} \rhd (\llbracket P_1 \rrbracket^{T} \cap \ldots \cap \llbracket P_n \rrbracket^{T}) \rhd (\llbracket Q_1 \rrbracket^{T} \cap \ldots \cap \llbracket Q_l \rrbracket^{T})\right)$$

A sequent is *valid* if any one of the predicates in the consequent is true in all the environments enriched by the declarations and satisfying all the predicates in the antecedent.

$$\llbracket D_1 \dagger \ldots \dagger D_m, \mid P_1, \ldots, P_n \vdash Q_1, \ldots, Q_l \rrbracket^{T} \;=\; {}^{\forall}(\llbracket D_1 \rrbracket^{T} ; \ldots ; \llbracket D_m \rrbracket^{T} \rhd (\llbracket P_1 \rrbracket^{T} \cap \ldots \cap \llbracket P_n \rrbracket^{T}))\,(\llbracket Q_1 \rrbracket^{T} \cup \ldots \cup \llbracket Q_l \rrbracket^{T})$$

A sequent is a *theorem* if it is valid in all environments.

F.3 Rules

The deductive system consists of a number of rules for manipulating sequents. A rule is of the form:

$$\mathsf{Rule} \;=\; \frac{\mathsf{Premisses}}{\mathsf{Conclusion}}\ [\uparrow\downarrow]\,[\mathsf{Name}]\,[(\mathsf{Proviso})]\ .$$

The premisses are a (possibly empty) list of sequents:

$$\mathsf{Premisses} \;=\; \mathsf{Sequent}\ldots\mathsf{Sequent}\ .$$

The conclusion is always a single sequent:

$$\mathsf{Conclusion} \;=\; \mathsf{Sequent}\ .$$

The Proviso is a decidable condition on the free variables and alphabets of the expressions in the rule. The Name usually has the form "$\exists\vdash$", or "$\vdash\exists$", the structure of which reflects the fact that there are rules for manipulating the operators of the logic, both on the left and on the right of the turnstile, respectively. The annotation $\uparrow\downarrow$ indicates that the rule can be applied in both directions.

A rule is *sound* if whenever it is applied to valid premisses, a valid conclusion results. This is defined in the semantics by saying that the set of environments supporting the premisses is a subset of those supporting the conclusion. The rule

$$\frac{S_1\ \ \ldots\ \ S_m}{Seq}\ [N](P)$$

is sound if and only if

$$P \;\Rightarrow\; \llbracket S_1 \rrbracket^{\mathcal{M}} \cap \ldots \cap \llbracket S_m \rrbracket^{\mathcal{M}} \subseteq \llbracket Seq \rrbracket^{\mathcal{M}}$$

The following meta-theorem holds for rules in the deductive system:

Theorem F.1 (Sequent-lifting)
The rule $\overline{D \mid \Psi \vdash \Phi}$ *is sound if and only if the sequent* $D \mid \Psi \vdash \Phi$ *is a theorem.*

This theorem states that a theorem can be deduced from no premisses.

In order to simplify the presentation of the deductive system the following lifting meta-theorem is used. It states that unchanging declarations and predicates can be added to a rule while maintaining soundness. An unchanging predicate or declaration is one that is in both the premiss and the conclusion.

Theorem F.2 (Rule-lifting)
If the inference rule $\dfrac{E \dagger D \mid \Psi \vdash \Phi}{E \dagger D' \mid \Psi' \vdash \Phi'}$ *is sound,*

then the rule $\dfrac{F \dagger E \dagger D \mid P, \Psi \vdash Q, \Phi}{F \dagger E \dagger D' \mid P, \Psi' \vdash Q, \Phi'}$ *is also sound,*

providing that $(\alpha D \cup \alpha D' \cup \alpha E) \cap (\phi P \cup \phi Q) = \varnothing$.

The rule-lifting theorem allows us to present the rules of the deductive system in a concise manner, by omitting any declarations and predicates which don't change.

The semantic-equivalences for substitution are given in tables in earlier sections. These tables state the semantic equality of various expressions. A theorem which permits the use of semantic-equivalences in proofs is the following.

Theorem F.3 (Semantic-equivalence-lifting) *Given the semantic-equivalences for predicates and declarations:*

$P \cong Q \qquad D \cong E,$

the following inference rules are sound:

$$\frac{E \mid Q \vdash}{D \mid P \vdash} \qquad \frac{E \vdash Q}{D \vdash P}$$

F.4 Proofs

Proofs in the deductive system proceed in the way that is usual for sequent calculi: proofs are developed *backwards*, starting from the sequent which is to be proved. A rule is applied, resulting in fresh sequents which must be proved. This process continues until there are no more sequents requiring proof, in which case the original sequent is now proved.

A completed proof may thus be represented as a tree, with the proved sequent as the root node, and every leaf node containing an empty list of sequents. However, if some of these lists in the leaves are non-empty, then the derivation tree is still useful, although it does not represent a proof, it represents a partial proof.

Theorem F.4 (Tree-squashing) *Suppose that we have the derivation tree:*

$$\frac{S_1 \quad \ldots \quad \dfrac{S_{i1} \quad \ldots \quad S_{im}}{S_i}\,[R_i](P_i) \quad \ldots \quad S_n}{Seq}\,[R](P)$$

where each of the rules R and R_i are sound rules, then the derived rule

$$\frac{S_1 \quad \ldots \quad S_{i1} \quad \ldots \quad S_{im} \quad \ldots \quad S_n}{Seq}\,[R'](P, P_i)$$

is also sound.

F.5 General Rules

F.5.1 Thin

The *thin rule* is used to discard unnecessary declarations and predicates:

$$\frac{\vdash}{D \mid P \vdash Q}\,(thin).$$

F.5.2 Assumption

The *assumption axiom* in is one way of completing a proof, since it leaves no premisses to be discharged; it states that for every formula p, the sequent $d \mid p \vdash p$ is valid:

$$\frac{}{D \mid P \vdash P} \; [assumption].$$

Notice that if we apply the Tree-squashing theorem to the assumption axiom preceded by the thin rule, we obtain the following:

$$\frac{}{D \mid D' \mid P, Q_1, \ldots, Q_m \vdash P, R_1, \ldots, R_n} .$$

Thus, the assumption axiom allows us to prove a sequent if any one of the consequent formulæ is present in the antecedent. This illustrates an important point about seqnent calculi: *every* formula on the left may be assumed in order to prove *at least one* formula ou the right.

F.5.3 Cut

The *cut rule* is used to structure proofs iuto lemmas; it permits the addition of hypotheses to the antecedent; these hypotheses may be discharged separately.

$$\frac{\vdash P \quad P \vdash}{\vdash} \; [cut].$$

It is the responsibility of the user of the cut rule to ensure that the well-typedness of the sequent is preserved by the addition of new predicates. New declarations can be cut in using an existentially quantified predicate.

F.6 Expressions

Two sets t and u are equal if and only if arbitrary members of t and u belong to u and t respectively:

$$\frac{x : t;\ y : u \vdash x \in u \land y \in t}{x : t;\ y : u \vdash t = u} \; \uparrow\downarrow \; [extension]$$

F.6.1 Set Extension

An element is a member of a set extension if and only if it is equal to one of the members of the extension.

$$\frac{\vdash t = u_1 \lor \ldots \lor t = u_n}{\vdash t \in \{u_1, \ldots, u_n\}} \; \uparrow\downarrow \; [extmem]$$

F.6.2 Set Comprehension

The element t is a member of the set comprehension $\{St \bullet u\}$ if and only if there is some situation in St which makes t equal u.

$$\frac{\vdash \exists\, St \bullet t = u}{\vdash t \in \{\ St \bullet u\ \}} \uparrow\downarrow\ {\scriptstyle [compre]}$$

$$\text{providing } \phi t \cap \alpha St = \varnothing$$

F.6.3 Power Set

An element is a member of a power set if and only if an arbitrary member of it is a member of the set.

$$\frac{y : t \vdash y \in u}{y : t \vdash t \in \mathbb{P}\ u} \uparrow\downarrow\ {\scriptstyle [powerset]}$$

F.6.4 Tuple

An element is equal to a tuple if and only if each of its projections is equal to the appropriate member of the tuple.

$$\frac{\vdash t.1 = u_1 \wedge \ldots \wedge t.n = u_n}{\vdash t = (u_1, \ldots, u_n)} \uparrow\downarrow\ {\scriptstyle [tuple]}$$

F.6.5 Cartesian Product

A tuple is a member of a Cartesian product if and only if each of its projections is a member of the respective member set of the product.

$$\frac{\vdash t.1 \in u_1 \wedge \ldots \wedge t.n \in u_n}{\vdash t \in (u_1 \times \ldots \times u_n)} \uparrow\downarrow\ {\scriptstyle [cartmem]}$$

F.6.6 Tuple Selection

The i^{th} projection of an explicit tuple is the i^{th} member of the tuple.

$$\frac{}{\vdash (u_1, \ldots, u_i, \ldots u_n).i = u_i}\ {\scriptstyle [tuplesel]}$$

F.6.7 Binding Extension

An element is equal to a binding extension if each of it selections is equal to the respective element of the binding.

$$\frac{\vdash b.n_1 = u_1 \land \ldots \land b.n_m = u_m}{\vdash b = \langle\!| n_1 \leadsto u_1, \ldots, n_m \leadsto u_m |\!\rangle} \uparrow\downarrow \; [binding]$$

F.6.8 Theta Expression

An explicit binding is equal to a theta expression if the decorated versions of the names in the binding equal the respective expressions.

$$\frac{\vdash n_1^q = u_1 \land \ldots \land n_m^q = u_m}{\vdash \langle\!| n_1 \leadsto u_1, \ldots, n_m \leadsto u_m |\!\rangle = \theta\ S\ ^q} \uparrow\downarrow \; [theta]$$

F.6.9 Schema Expression

A binding is a member of a schema expression if and only if the schema is true following the substitution of the binding.

$$\frac{\vdash b \odot S}{\vdash b \in S} \uparrow\downarrow \; [schemaexp]$$

F.6.10 Binding Selection

The projection of the name n_i from an explicit binding is the element to which the name is mapped.

$$\frac{}{\vdash \langle\!| n_1 \leadsto u_1, \ldots, n_i \leadsto u_i, \ldots, n_m \leadsto u_m |\!\rangle .n_i = u_i} \; [tuplesel]$$

F.7 Predicates

F.7.1 Equality

All expressions are equal to themselves.

$$\frac{}{\vdash x = x} \; [reflection]$$

F.7.2 Truth

$$\frac{}{\vdash \mathsf{true}}\ [\mathit{truth}]$$

F.7.3 Falsehood

$$\frac{}{\mathsf{false} \vdash}\ [\mathit{contradiction}]$$

F.7.4 Negation

If the predicate $\neg p$ is in the antecedent then one way to proceed is by proving a contradiction i.e. that p is true.

$$\frac{\vdash p}{\neg p \vdash}\ \uparrow\downarrow\ [\neg\vdash]$$

If the predicate $\neg p$ is in the consequent then if p does not hold then there is a proof, so it can be assumed that it does hold:

$$\frac{p \vdash}{\vdash \neg p}\ \uparrow\downarrow\ [\vdash\neg]$$

F.7.5 Conjunction

The conjunction of two predicates in the antecedent is the same as having them both in the predicate list:

$$\frac{p, q \vdash}{p \wedge q \vdash}\ \uparrow\downarrow\ [\wedge\vdash]$$

The conjunction of two predicates can be proved only if both of the predicates can be proved.

$$\frac{\vdash p \quad \vdash q}{\vdash p \wedge q}\ [\vdash\wedge]$$

F.7.6 Disjunction

Given a disjunction of two predicates in the antecedent it is necessary to be able to complete the proof with eitherpredicate in the assumption.

$$\frac{p \vdash \quad q \vdash}{p \lor q \vdash} \; [\lor\vdash]$$

A disjunction of two predicates in the consequent is the same as having them both in the consequent.

$$\frac{\vdash p, q}{\vdash p \lor q} \; \uparrow\downarrow \; [\vdash\lor]$$

F.7.7 Implication

$$\frac{p \Rightarrow q \vdash p \quad q \vdash}{p \Rightarrow q \vdash} \; [\Rightarrow\vdash]$$

$$\frac{p \vdash q}{\vdash p \Rightarrow q} \; \uparrow\downarrow \; [\vdash\Rightarrow]$$

F.7.8 Equivalence

$$\frac{p \Rightarrow q, q \Rightarrow p \vdash}{p \Leftrightarrow q \vdash} \; \uparrow\downarrow \; [\Leftrightarrow\vdash]$$

$$\frac{\vdash p \Rightarrow q \quad \vdash q \Rightarrow p}{\vdash p \Leftrightarrow q} \; [\vdash\Leftrightarrow]$$

F.7.9 Universal Quantification

$$\frac{b \in \{St], \forall St \bullet p, b \odot p \vdash}{b \in \{St], \forall St \bullet p \vdash} \; \uparrow\downarrow \; (\forall \vdash)$$

If we have to prove the predicate $\forall d \mid p \bullet q$ then it can be assumed that the variables in d are arbitrary, and that they satisfy the property of $d \mid p$, leaving the predicate q to be proved.

$$\frac{d \mid p \vdash q}{\vdash \forall d \mid p \bullet q} \; \uparrow\downarrow \; (\vdash \forall)$$

F.7.10 Existential Quantification

Suppose that we have the single antecedent $\exists d \mid p \bullet q$; that is, we know that there is some way of constructing the variables in d such that the property of d and the predicates p and q are satisfied.

Although we may not know such a construction, we can give arbitrary names to the variables of d to stand for an arbitrary construction satisfying d, p and q. If we take as our new assumption $d \mid p \wedge q$, the variables of d are indeed arbitrary, since they cannot be global ones, and no other local names are in the antecedent.

$$\frac{d \mid p \wedge q \vdash}{\exists\, d \mid p \bullet q \vdash}\ (\exists \vdash)$$

Suppose that we have the consequent $\exists\, St \bullet p$, and suppose that we know a binding that satisfies the property of St. One way forward is to prove that this binding also satisfies the predicate p. It is convenient to retain the consequent, in case we wish to try to prove that other bindings satisfy p.

$$\frac{b \in [St] \vdash \exists\, St \bullet p, b \odot p}{b \in [St] \vdash \exists\, S \bullet p}\ (\vdash \exists)$$

F.7.11 Substitution

$$\frac{s = t, ⦃\, x \leadsto t\, ⦄ \odot p \vdash}{s = t, ⦃\, x \leadsto s\, ⦄ \odot p \vdash}\ (\mathit{Leibniz})$$

F.8 Schemas

The schema rules are based of the definitions of schema predicates and hence follow very closely the rules for predicates:

F.8.1 Schema Construction

$$\frac{[d] \wedge p \vdash}{[d \mid p] \vdash}\ [[|]\vdash]$$

$$\frac{\vdash [d] \wedge p}{\vdash [d \mid p]}\ [\vdash[|]]$$

F.8.2 Schema Negation

$$\frac{\vdash [S]}{[\neg S] \vdash}\ [[\neg]\vdash]$$

$$\frac{[S] \vdash}{\vdash [\neg S]}\ [\vdash[\neg]]$$

F.8.3 Schema Disjunction

$$\frac{[S] \vdash \quad [T] \vdash}{[S \vee T] \vdash}\ [[\vee]\vdash]$$

$$\frac{\vdash [S], [T]}{\vdash [S \vee T]}\ [\vdash[\vee]]$$

F.8.4 Schema Conjunction

$$\frac{[S], [T] \vdash}{[S \wedge T] \vdash}\ [[\wedge]\vdash]$$

$$\frac{\vdash [S] \quad \vdash [T]}{\vdash [S \wedge T]}\ [\vdash[\wedge]]$$

F.8.5 Schema Implication

$$\frac{[S \Rightarrow T] \vdash [S] \quad [T] \vdash}{[S \Rightarrow T] \vdash}\ [[\Rightarrow]\vdash]$$

$$\frac{[S] \vdash [T]}{\vdash [S \Rightarrow T]}\ [\vdash[\Rightarrow]]$$

F.8.6 Schema Equivalence

$$\frac{[S \Rightarrow T], [T \Rightarrow S] \vdash}{[S \Leftrightarrow T] \vdash}\ [[\Leftrightarrow]\vdash]$$

$$\frac{\vdash [S \Rightarrow T] \quad \vdash [S \Rightarrow T]}{\vdash [S \Leftrightarrow T]}\ [\vdash[\Leftrightarrow]]$$

Note: There are more rules to be added here.

F.9 Declarations

$$\frac{D \mid [D] \vdash}{D \mid\vdash}\ [d]$$

F.9.1 Simple Declaration

$$\frac{n_1 \in s \land \ldots \land n_m \in s \vdash}{[n_1, \ldots, n_m : s] \vdash} \; [[n:s]\vdash]$$

$$\frac{\vdash n_1 \in s \land \ldots \land n_m \in s}{\vdash [n_1, \ldots, n_m : s]} \; [\vdash[n:s]]$$

F.9.2 Compound Declaration

$$\frac{[D_1] \land [D_1] \vdash}{[D_1;\ D_2] \vdash} \; [[D;D]\vdash]$$

$$\frac{\vdash [D_1] \land [D_2]}{\vdash [D_1;\ D_2]} \; [\vdash[D;D]]$$

F.10 Definitions

F.10.1 Axiomatic Definition

Providing the specification contains the declaration

$$\begin{array}{|l} D \\ \hline P \end{array}$$

we have the inference rule

$$\frac{[D \mid P] \vdash}{\vdash} \; (AxiomDef)$$

F.10.2 Generic Definition

F.10.3 Schema Definition

Providing the specification contains the declaration

$S[X_1, \ldots, X_m] \mathrel{\widehat=} T$

we have the inference rule

$$\frac{S[t_1, \ldots, t_m] = \{\!| X_1 \rightsquigarrow t_1, \ldots, X_m \rightsquigarrow t_m |\!\} \odot T \vdash}{\vdash} \; (\text{SchemaGenDef})$$

F.10.4 Constraint

Providing the specification contains the constraint

P

we have the inference rule

$$\frac{P \vdash}{\vdash} \text{ (Constraint)}$$

References

[1] Abrial, J-R., "A Course on System Specification," Lecture Notes, Programming Research Group, University of Oxford, 1981.

[2] Bowen, J., "Select Z Bibliography," in J. E. Nicholls (ed), *Z User Workshop, York 1991*, Proceedings of the Sixth Annual Z User Meeting, Springer-Verlag, 1992.

[3] Brien, S.M., Gardiner, P.H.B., Lupton, P.J., Woodcock, J.C.P., "A Semantics for Z," in preparation, 1992.

[4] BSI Standard **BS 0** : Part 1 : 1981, *A standard for standards. Part 1. Guide to general principles of standardization*, British Standards Institution, 1991.

[5] BSI Standard **BS 6154**, *Method of defining syntactic metalanguage*, British Standards Institution, 1981.

[6] Enderton, H.B., *Elements of set theory*, Academic Press, 1977.

[7] Gardiner, P.H.B., Lupton, P.J., Woodcock, J.C.P., "A simpler semantics for Z," in J. E. Nicholls (ed), *Z User Workshop, Oxford 1990*, Proceedings of the Fifth Annual Z User Meeting, Springer-Verlag, 1991.

[8] Goldfarb, C. F., *The SGML Handbook*, Clarendon Press, Oxford, 1990.

[9] Hamilton, A.G., *Numbers, sets and axioms*, Cambridge University Press, 1982.

[10] Hayes, I. J., (ed.), *Specification Case Studies*, Prentice-Hall International, 1987.

[11] ISO (International Organization for Standardization), **ISO 8879-1986 (E)** *Information Processing – Text and Office systems – Standard Generalized Markup Language (SGML)*, Geneva: ISO, 1986.

[12] Jones, C. B., *Software Development–A Rigorous Approach*, Prentice-Hall International, 1980.

[13] Jones, C. B., *Systematic Software Development using VDM*, Prentice-Hall International, 1986.

[14] King, S., Sørensen, I. H., Woodcock, J.C.P., "Z: Grammar and Concrete and Abstract Syntaxes (Version 2.0)", Technical Monograph PRG-68, Programming Research Group, University of Oxford, 1988.

[15] Morgan, C. C., "Schemas in Z: a Preliminary Reference Manual," Programming Research Group, University of Oxford, 1984.

[16] Nicholls, J. E., "Domains of application for formal methods," in J. E. Nicholls (ed), *Z User Workshop, York 1991*, Proceedings of the Sixth Annual Z User Meeting, Springer-Verlag, 1992.

[17] Sennett, C. T., "Syntax and Lexis of the Specification Language Z," RSRE Memorandum No. 4367, 1990.

[18] Sørensen, I. H., "A Specification Language," in *Program Specification* (J. Staunrup, ed.), Lecture Notes in Computer Science, vol. 134, Springer-Verlag, 1982.

[19] Spivey, J. M., *Understanding Z: a specification language and its formal semantics*, Cambridge University Press, 1988.

REFERENCES

[20] Spivey, J. M., *The Z notation – a reference manual*, Prentice-Hall International, 1989. *(2nd edition, 1992).*

[21] Stoy, J.E., *Denotational semantics: the Scott-Strachey approach to programming language theory*, MIT Press, 1977.

[22] Sufrin, B. A., "Formal Specification: Notation and Examples," in *Tools and Notations for Program Construction* (D. Néel, ed.). Cambridge University Press, 1981.

[23] Sufrin, B. A., (ed.), "Z Handbook, Draft 1.1," Programming Research Group, University of Oxford, 1986.

[24] Sperberg-McQueen, C.M., Burnard, L., *Text Encoding Initiative: Guidelines for the encoding and interchange of machine-readable texts*, Draft Version 1.0. Chicago and Oxford, 1990.

[25] Woodcock, J. C. P., "Structuring Specifications in Z," *Software Engineering Journal*, Vol 4, No 1 (January 1989).

[26] Woodcock, J.C.P., Brien, S.M., "$\mathcal{W}$: a logic for Z," in J. E. Nicholls (ed), *Z User Workshop, York 1991*, Proceedings of the Sixth Annual Z User Meeting. Springer-Verlag, 1992.