Z Notation

Version 1.2

14th September 1995

Prepared by members of the Z Standards Panel

BSI Panel IST/5/-/19/2 (Z Notation)
ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z)

Project editor: John Nicholls

This is a major revision of the Z Standard, replacing all previous versions. It is an updated version of the document presented at the 29th Meeting of the Z Standards Panel held on the 6th September 1995.

Published in the United Kingdom

Document history:

Version 0.1 (19th March 1991): First Version of ZIP Deliverable D1.3.1.

Version 0.2 (15th May 1991): Prepared for a meeting of the ZIP Standards Review Committee on 10th May, 1991.

Version 0.3 (25th November 1991): Produced for a meeting of the Standards Review Committee in December 1991.

Version 0.4 (9th December 1991): Substantially the same as Version 0.3.

Version 0.5 (20th March 1992): Revision and extension of Version 0.4.

Version 0.6 (21st October 1992): Major revision, in which sections containing the language description are restructured and updated, and all other sections revised.

Version 1.0 (30th November 1992): A version prepared for distribution at the 7th Z User Meeting. Adopted as Working Draft of the Z Standards Panel.

Version 1.1 (30th June 1995): Special issue for review by the Z Standards Panel.

Acknowledgement

Preparation of earlier versions of this document was supported as part of the ZIP project. *ZIP — A unification initiative for Z Standards, Methods and Tools* was partially funded by the Department of Trade and Industry and the Science and Engineering Council under their joint Information Engineering Advanced Technology Programme.

The work of the Z Standards Panel is supported by:

CESG Cheltenham
DRA Malvern
IBM United Kingdom Laboratories
US Department of Defense

Contributors

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

Project editor

John Nicholls

Section authors and editors for this version

Randolph Johnson
Jonathan Hammond
Peter Lupton
Andrew Martin
John Nicholls
Susan Stepney
John Wordsworth

Original text for this version by

Rob Arthan
Stephen Brien
Will Harwood
Trevor King
Pete Steggles
Jim Woodcock

Contributors to this and previous versions

Paul Gardiner
Roger Jones
Jon Hall
Ian Hayes
Steve King
Mark Saaltink
Chris Sennett
Ib Sørensen
Sam Valentine
...

Other 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 209). 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 Standards Panel

Development of the Z Base Standard has benefited from contributions and reviews by the *Z Standards Panel*, previously known as the *Z Standards Review Committee.*

The following list shows the current members of the Z Standards Panel, together with their alternates or deputies.

Derek Andrews	University of Leicester UK
Rob Arthan	ICL Winnersh UK
Peter Baumann	University of Zurich Switzerland
Cinzia Bonini	INTECS Pisa Italy
Stephen Brien	Oxford University PRG UK
Colin Champion	CESG Cheltenham UK
John Dawes	ICL Reading UK
Susan Gerhart	University of Houston USA
Jon Hall	University of York UK
Jonathan Hammond	Praxis plc Bath UK
Will Harwood	Imperial Software Technology Cambridge UK
Ian Hayes	University of Queensland Australia
Randolph Johnson	DoD Fort Meade USA
Roger Jones	ICL Winnersh UK
Steve King	Oxford University PRG UK
Trevor King	Praxis plc Bath UK (up to Meeting 27)
Peter Lupton	IBM United Kingdom Laboratories Hursley UK
Andrew Martin	Oxford University PRG UK
Peter Mataga	AT & T Bell Laboratories Naperville USA
Silvio Meira	Federal University of Pernambuco Brazil
Akira Nakamura	Toshiba Corporation Kawasaki Japan
John Nicholls	Oxford University PRG UK *(Convener)*
Colin Parker	British Aerospace Warton UK
Jan Peleska	DST Kiel Germany
Brian Ritchie	Rutherford Appleton Laboratories Chilton UK
Mark Saaltink	ORA Ottawa Canada
Mayer Schwartz	Tektronix Oregon USA
Jane Sinclair	Open University UK
Alf Smith	DRA Malvern UK
Makoto Someya	Unisys Nihon Tokyo Japan
Pete Steggles	Imperial Software Technology Cambridge UK
Susan Stepney	Logica Cambridge UK
Mitsukazu Uchiyama	FDT-SWG Japan
Sam Valentine	University of Brighton UK
Jim Woodcock	Oxford University PRG UK
John Wordsworth	IBM United Kingdom Laboratories Hursley UK
Pete Young	British Telecom UK

Contents

Foreword **ix**

Introduction **x**

1 Scope **1**

2 Conformance **2**

3 Normative references **3**

4 Semantic metalanguage **4**
- 4.1 Introduction 4
- 4.2 Definitions and declarations 5
- 4.3 Sets 6
- 4.4 Tuples and products 8
- 4.5 Relations 9
- 4.6 Functions 10
- 4.7 Set constructors as relations 11
- 4.8 Compatible functions. 12
- 4.9 Diagonal and projection 13
- 4.10 Pointwise product 13
- 4.11 Relational tuple 14
- 4.12 Promoted application 15

5 Semantic universe **16**
- 5.1 Introduction 16
- 5.2 Names and types 16
- 5.3 Values in Z 17
- 5.4 Elements in Z 20
- 5.5 Generics 21
- 5.6 Environments 23
- 5.7 Generic lifting 24

6 Language description **27**
- 6.1 Introduction 27
- 6.2 Abstract syntax 28
- 6.3 Concrete form, representation and transformation 28
- 6.4 Type 30
- 6.5 Meaning 31
- 6.6 Value 31
- 6.7 Free variables 32
- 6.8 Alphabet 33
- 6.9 Substitution 33

7 Expression **35**
- 7.1 Introduction 35
- 7.2 Identifier 38
- 7.3 Generic instantiation 39
- 7.4 Number literal 41
- 7.5 Set extension 42
- 7.6 Set comprehension 43
- 7.7 Power set 45

7.8 Tuple 46
7.9 Cartesian product 47
7.10 Tuple selection 48
7.11 Binding extension 49
7.12 Theta expression 50
7.13 Schema expression 52
7.14 Binding selection 53
7.15 Application 54
7.16 Definite description 56
7.17 Conditional expression 57
7.18 Substitution 58

8 Predicate **60**
8.1 Introduction 60
8.2 Equality 62
8.3 Membership 63
8.4 Truth literal 64
8.5 False literal 65
8.6 Negation 66
8.7 Disjunction 67
8.8 Conjunction 68
8.9 Implication 69
8.10 Equivalence 70
8.11 Universal quantification 71
8.12 Existential quantification 72
8.13 Unique existential quantification 73
8.14 Schema predicate 74
8.15 Substitution 75

9 Schema **77**
9.1 Introduction 77
9.2 Schema designator 80
9.3 Generic schema designator 81
9.4 Simple schema 82
9.5 Schema construction 83
9.6 Schema negation 84
9.7 Schema disjunction 85
9.8 Schema conjunction 86
9.9 Schema implication 87
9.10 Schema equivalence 88
9.11 Schema projection 89
9.12 Schema hiding 90
9.13 Schema universal quantification 91
9.14 Schema existential quantification 92
9.15 Schema unique existential quantification 93
9.16 Schema renaming 94
9.17 Substituted schema 95

10 Section **96**
10.1 Introduction 96

11 Paragraph **97**
11.1 Introduction 97

11.2 Given set . . . 98
11.3 Global declaration . . . 99
11.4 Constraint . . . 100
11.5 Generic declaration . . . 101
11.6 Global definition . . . 103
11.7 Generic definition . . . 104
11.8 Free types . . . 105
11.9 Conjecture . . . 106

12 Specification **107**
12.1 Introduction . . . 107

A Abstract syntax – Normative Annex **109**
A.1 Introduction . . . 109
A.2 Metalanguage for abstract syntax . . . 109
A.3 Specification . . . 111
A.4 Section . . . 111
A.5 Paragraph . . . 111
A.6 Schema . . . 112
A.7 Predicate . . . 113
A.8 Expression . . . 114
A.9 Name . . . 114

B Concrete syntax – Normative Annex **115**
B.1 Introduction . . . 115
B.2 Metalanguage for concrete syntax . . . 115
B.3 Specification . . . 116
B.4 Paragraph . . . 116
B.5 Predicate . . . 117
B.6 Expression and schema expression . . . 118
B.7 Lists . . . 120
B.8 Operator templates . . . 121

C Lexis – Normative Annex **128**
C.1 Introduction . . . 128
C.2 Soft newlines . . . 128
C.3 Tokens . . . 129
C.4 Glyphs . . . 132
C.5 Toolkit glyphs . . . 138

D Mathematical toolkit – Normative Annex **140**
D.1 Sets . . . 141
D.2 Relations . . . 146
D.3 Functions . . . 154
D.4 Numbers and finiteness . . . 158
D.5 Sequences . . . 165
D.6 Bags . . . 172

E Interchange format – Normative Annex **175**
E.1 Introduction . . . 175
E.2 Scope of the Interchange Format . . . 175
E.3 Introduction to SGML . . . 176
E.4 Definition of the Interchange Format . . . 180
E.5 Examples . . . 185

F The logical theory of Z – Normative Annex **188**
F.1 Preamble 188
F.2 Meta-language 188
F.3 Free variables and alphabets 193
F.4 Substitution 196
F.5 Type inference 200
F.6 Inference rules 204

G References – Informative Annex **209**

Foreword

Notes on this section of the Z Standard

Section title: Foreword
Section editor: John Nicholls
Source file: part of front.tex
Most recent update: 7th June 1995
Formatted: 14th September 1995

This is a major revision of the Z Standard, replacing all previous versions.

This is the current version of the Z Standard being developed as a BSI and ISO standard. It is the Working Draft (WD) of the Z Standards Panel, BSI Panel IST/5/-/19/2: *Z Notation*), ISO Panel SC22/WG19 (Rapporteur Group for Z).

□

Introduction

Notes on this section of the Z Standard

Section title: Introduction
Section editor: John Nicholls
Source file: intro.tex
Most recent update: 13 September 1995
Formatted: 14th September 1995

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.

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 [20].

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 [3].

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.

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.

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 been used in significant case studies or development projects.

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

Aims of standardisation

The Z standard supports the following general aims of standardisation as listed in the British Standards Institution *Standard for Standards* [6]:

> 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.

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.

□

1 Scope

Notes on this section of the Z Standard

Section title: Scope
Section editor: John Nicholls
Contributions by:
Source file: scope.tex
Most recent update: 31 August 1995
Formatted: 14th September 1995

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.

□

2 Conformance

Notes on this section of the Z Standard

Section title: Conformance
Section editor: John Nicholls
Source file: conform.tex
Most recent update: 16 January 1995
Formatted: 14th September 1995

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.

□

3 Normative references

Notes on this section of the Z Standard

Section title: Normative references
Section editor: John Nicholls
Source file: normref.tex
Most recent update: 29th June 1995
Formatted: 14th September 1995

BSI6154 BSI Standard **BS 6154**, *Method of defining syntactic metalanguage*, British Standards Institution, 1981.

ISO8879 ISO (International Organization for Standardization), **ISO 8879-1986 (E)** *Information Processing – Text and Office systems – Standard Generalized Markup Language (SGML)*, Geneva: ISO, 1986.

□

4 Semantic metalanguage

Notes on this section of the Z Standard

Section title: Semantic metalanguage
Section editor: Randolph Johnson
Contributions by: Stephen Brien, Randolph Johnson, John Nicholls, Jim Woodcock, ... *(others to be added)*
Source file: math.tex
Most recent update: 13th September 1995
Formatted: 14th September 1995

4.1 Introduction

This is the first of two chapters describing the mathematical framework used in the definition of Z. The chapter includes:

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

Many of the symbols used in this chapter are derived from conventional mathematics and are defined informally. Throughout the standard, the mathematical treatment is based on Zermelo-Fraenkel (ZF) set theory. An introduction to ZF theory can be found in text books such as Enderton [8] or Hamilton [11].

In addition to conventional mathematical symbols, special symbols are introduced which allow concise semantic definitions to be written. Where these have meanings similar to those of Z, Z-like symbols are used. Definitions of new symbols are given 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 mathematics based on classical (i.e. untyped) set theory.

Naming conventions. The following naming conventions are used:

- upper-case letters A, B, C (sometimes with subscripts) denote sets;
- upper-case letters R, S, T (sometimes with subscripts) denote relations;
- lower-case letters a, b, c (sometimes with subscripts) denote members of sets (which may also be sets themselves).

Commuting diagrams. In several of the following descriptions *commuting diagrams* are used to illustrate relationships between the set constructors being defined. Commuting diagrams are directed graphs whose nodes are labeled with sets and whose edges are labeled with relations. Each directed path in the diagram has an associated relation which is the composition of the relations labeling its constituent edges. A diagram is said to *commute* when any two directed paths from one node to another have the same associated relation.

Elision An expression such as $a_1, \ldots, a_n$ indicates that intermediate members in the finite list are elided. Whenever such an expression is used, the minimum value that n is allowed to take (usually 0 or 1) will be stated. An expression such as $A^1 \cup A^2 \cup \ldots$ indicates that all (finite) values of the superscript are to be included.

4.2 Definitions and declarations

Variables and notations are introduced and named as follows:

Table 1: Declarations and definitions

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

Note: A declaration of a in the form $a : A$ introduces the object a and states that it is a member of the set A. Care has been taken to ensure that when such a declaration is used, the set A is non-empty.

4.3 Sets

The following sets are predefined:

Table 2: Predefined sets

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

Relationships between sets and their members are written as follows:

Table 3: Relationships between sets and members

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

4.3.1 Set constructors

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

Table 4: Set constructors

Name	Form	Description
set extension	$\{a_1, \ldots, a_n\}$	the set comprising $a_1, \ldots, a_n$; if $n = 0$, the set is the empty set; if $n = 1$, the set is a *singleton* set
union	$A \cup B$	the set comprising all the members of A and all the members of B
intersection	$A \cap B$	the set comprising the members common to A and B
set difference	$A \setminus B$	the set comprising the members of A that are not members of B
power set	$\mathbb{P}\, A$	the set of all subsets of A
finite power set	$\mathbb{F}\, A$	the set of all finite subsets of A

4.4 Tuples and products

The following constructors define tuples and products:

Table 5: Tuples and products

Name	Form	Definition
tuple	$(a_1, \ldots, a_n)$	ordered list of the elements $a_1, \ldots, a_n$, where $n \geq 1$
Cartesian product	$A_1 \times \ldots \times A_n$	the set of tuples $(a_1, \ldots, a_n)$ such that $a_1 \in A_1$ and … and $a_n \in A_n$
enumerated product	A^n	the set of tuples $(a_1, \ldots, a_n)$ such that $a_1, \ldots, a_n \in A$
iterated product	A^+	$A^1 \ \cup \ A^2 \ \cup \ A^3 \ \cup \ \ldots$

Note: It is important in the metalanguage that the sets $\{A^1, A^2, A^3, \ldots\}$ form a disjoint collection. Under the most common construction of tuples, this need not be the case. However, it is true if, for example, we view a tuple as a finite sequence so that $(a_1, \ldots, a_n)$ is a function from $\{1, 2, \ldots, n\}$ into A.

4.5 Relations

The following are defined:

Table 6: Relations

Name	Form	Definition
relations	$A \leftrightarrow B$	$\mathbb{P}(A \times B)$
identity relation	I_A	$(a, b) \in I_A \Leftrightarrow a = b \wedge a \in A$
domain	$\mathrm{dom}\, R$	$a \in \mathrm{dom}\, R \Leftrightarrow \exists\, b \bullet (a, b) \in R$
range	$\mathrm{ran}\, R$	$b \in \mathrm{ran}\, R \Leftrightarrow \exists\, a \bullet (a, b) \in R$
converse	R^{-1}	$(a, b) \in R^{-1} \Leftrightarrow (b, a) \in R$
composition	$R \mathbin{;} S$	$(a, b) \in R \mathbin{;} S \Leftrightarrow$ $\exists\, c \bullet (a, c) \in R \wedge (c, b) \in S$
range restriction	$R \rhd A$	$R \mathbin{;} I_A$
range anti-restriction	$R \unrhd A$	$R \rhd (\mathrm{ran}\, R \setminus A)$
domain restriction	$A \lhd R$	$I_A \mathbin{;} R$
domain anti-restriction	$A \unlhd R$	$(\mathrm{dom}\, R \setminus A) \lhd R$

Note: The composition operator binds more tightly than the set constructors.

Note: The subscript A may be omitted if the domain of the identity relation can be determined from the context.

4.6 Functions

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

Table 7: Functions

Name	Form	Description or definition
partial functions	$A \mathrel{+\!\!\!\!\rightarrow} B$	the set of functions from A into B whose domains are subsets of A
total functions	$A \longrightarrow B$	the set of functions from A into B whose domains are the whole of A
total injections	$A \rightarrowtail B$	the set of total functions from A into B which are one-to-one
total surjections	$A \twoheadrightarrow B$	the set of total functions from A into B whose ranges are B
bijections	$A \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} B$	$A \rightarrowtail B \ \cap \ A \twoheadrightarrow B$
finite partial functions	$A \mathrel{+\!\!\!\!+\!\!\!\!\rightarrow} B$	$A \mathrel{+\!\!\!\!\rightarrow} B \ \cap \ \mathbb{F}(A \times B)$

Table 8: Function constructors

Name	Form	Description or definition
constant function	a°_{A}	maps all members in the set A to a $(b, a) \in a^{\circ}_{A} \Leftrightarrow b \in A$
relational image	${}^{\exists}(R)$	$(A, B) \in {}^{\exists}(R) \Leftrightarrow B = \mathrm{ran}(A \lhd R)$
singleton image	${}^{\Lambda}(R)$	$(a, B) \in {}^{\Lambda}(R) \Leftrightarrow B = \mathrm{ran}(\{a\} \lhd R)$

Note: The subscript A may be omitted if the domain can be determined from the context.

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

4.7 Set constructors as relations

Membership, union, intersection etc. are not sets, and therefore, *a fortiori*, not relations. However, the restriction of any of these to the members or subsets etc. of a particular set A, does indeed yield a relation. In general, the relation determined, e.g., by the membership function, will depend on the set A; however, it is convenient to use a notation which supresses this dependence when A is clear from the context of use. The notations used are defined in the following table.

Table 9: Set constructors defined as relations

Name	Symbol	Domain	Range	Definition
union	$(\cup)$	$(\mathbb{P}\, A)^2$	$\mathbb{P}\, A$	$((a_1, a_2), b) \in (\cup) \;\Leftrightarrow\; b = a_1 \cup a_2$
intersection	$(\cap)$	$(\mathbb{P}\, A)^2$	$\mathbb{P}\, A$	$((a_1, a_2), b) \in (\cap) \;\Leftrightarrow\; b = a_1 \cap a_2$
set difference	$(\setminus)$	$(\mathbb{P}\, A)^2$	$\mathbb{P}\, A$	$((a_1, a_2), b) \in (\setminus) \;\Leftrightarrow\; b = a_1 \setminus a_2$
containment	$(\supseteq)$	$\mathbb{P}\, A$	$\mathbb{P}\, A$	$(a, b) \in (\supseteq) \;\Leftrightarrow\; b \subseteq a$
member	$(\ni)$	$\mathbb{P}\, A$	A	$(a, b) \in (\ni) \;\Leftrightarrow\; b \in a$
singleton set	$\{-\}$	A	$\mathbb{P}\, A$	$(a, b) \in \{-\} \;\Leftrightarrow\; b = \{a\}$
tuple set	$\{..\}$	A^+	$\mathbb{P}\, A$	$((a_1, \ldots, a_n), b) \in \{..\} \;\Leftrightarrow$ $b = \{a_1, \ldots, a_n\}$
power	$(\mathbb{P})$	$\mathbb{P}\, A$	$\mathbb{P}\,\mathbb{P}\, A$	$(a, b) \in (\mathbb{P}) \;\Leftrightarrow\; b = \mathbb{P}\, a$
relational override	$(\oplus)$	$(A \longleftrightarrow B)^2$	$A \longleftrightarrow B$	$((R_1, R_2), S) \in (\oplus) \;\Leftrightarrow$ $S = (\mathrm{dom}\, R_2 ⩤ R_1) \cup R_2$
Cartesian product	$(\times)$	$(\mathbb{P}\, A)^+$	$\mathbb{P}(A^+)$	$((a_1, \ldots, a_n), b) \in (\times) \;\Leftrightarrow$ $b = a_1 \times \ldots \times a_n$
indexed product	$\mathcal{X}$	$A ⇸ \mathbb{P}\, B$	$\mathbb{P}(A ⇸ B)$	$(a, b) \in (\mathcal{X} ⨾ (\ni)) \Leftrightarrow b \subseteq a ⨾ (\ni)$ and $\mathrm{dom}\, a = \mathrm{dom}\, b$.

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

The following diagram shows commuting properties of relational constructors:

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

4.8 Compatible functions.

Two functions are said to be *compatible* if their union is also a function. That is, their values agree whenever their domains overlap.

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

$$\mathcal{C}_{AB} \mathrel{\widehat{=}} \mathrm{dom}((\cup) \rhd (A \mathrel{⇸} B))$$

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 10: Constructors on compatible functions

Name	Symbol	Definition
functional union	$\sqcup_{AB}$	$\mathcal{C}_{AB} \lhd (\cup)$
functional intersection	$\sqcap_{AB}$	$\mathcal{C}_{AB} \lhd (\cap)$
functional difference	$\leftharpoonup_{AB}$	$\mathcal{C}_{AB} \lhd (\setminus)$

4.9 Diagonal and projection

The following are defined:

Table 11: Diagonal and projection operators

Name	Symbol	Domain	Range	Definition
diagonal	Δ_n	A	A^n	$(a, (a_1, \ldots, a_n)) \in \Delta_n$ $\Leftrightarrow\ a_1 = a \wedge \ldots \wedge a_n = a,$ where $n \geq 1$
projection	π_i	$A_1 \times \ldots \times A_i \times \ldots \times A_n$	A_i	$((a_1, \ldots, a_i, \ldots, a_n), a) \in \pi_i$ $\Leftrightarrow\ a = a_i$, where $1 \leq i \leq n$

4.10 Pointwise product

The pointwise product $R_1 \otimes \ldots \otimes R_n$ is a relation from the Cartesian product of the domains of $R_1, \ldots, R_n$ to the Cartesian product of their ranges.

Table 12: Pointwise product constructors

Name	Form	Definition
pointwise product	$R_1 \otimes \ldots \otimes R_n$	$((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \in R_1 \otimes \ldots \otimes R_n$ $\Leftrightarrow\ (a_1, b_1) \in R_1 \wedge \ldots \wedge (a_n, b_n) \in R_n$
iterated pointwise product	$R^{\oplus}$	$(\Delta_1^{-1} \fatsemi R \fatsemi \Delta_1) \cup (R \otimes R)$ $\cup\ (R \otimes R \otimes R) \cup \ldots$

The following diagram illustrates properties of the product constructors:

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

4.11 Relational tuple

If A is the intersection of the domains of the relations $R_1, \ldots, R_n$, then the relational tuple $\langle R_1, \ldots, R_n \rangle$ is a relation from A to the Cartesian product of their ranges. This is defined in terms of the relational product operator and the diagonal operator Δ_n.

Table 13: Tuple constructor

Name	Form	Definition
relational tuple	$\langle R_1, \ldots, R_n \rangle$	$\Delta_n \fatsemi (R_1 \otimes \ldots \otimes R_n)$, where $n \geq 1$

If the relations $R_1, \ldots, R_n$ each have domain A and have range B_i, respectively, then the following diagram shows the relationship between relational tuple $\langle R_1, \ldots, R_n \rangle$ and projection:

$$\begin{tikzcd}
& B_i \\
B_1 \times \ldots \times B_n \arrow[ur, "\pi_i"] & \\
& A \arrow[ul, "\langle R_1, \ldots, R_n \rangle"] \arrow[uu, "R_i"']
\end{tikzcd}$$

Table 14: Relational tuple operator

Name	Symbol	Domain	Range	Definition
relational tupling	$\diamond$	$(A \longleftrightarrow B_1) \times (A \longleftrightarrow B_2)$	$A \longleftrightarrow (B_1 \times B_2)$	$((R_1, R_2), S) \in \diamond$ $\Leftrightarrow S = \langle R_1, R_2 \rangle$

4.12 Promoted application

In order to avoid generating potentially undefined terms, there is no function application in the meta-language (it is used only in explanatory notes). For the definition of function application in Z, it is necessary to define a form of promoted application. For any given a, the apply-to-a function takes as its argument a function and has as its result the application of that function to the element a. This function can be generalised to the *promoted application* operator $(R \bullet T)$, which is the relational analogue of the S combinator in combinatory logic.

Table 15: Promoted application

Name	Form	Domain	Range	Definition
apply-to-a	$(\_a)$	$A \longleftrightarrow B$	B	$((\ni) \cap (a^\circ \mathbin{⨾} \pi_1^{-1})) \mathbin{⨾} \pi_2$
promoted application	$R \bullet S$	$A \longleftrightarrow (B \longleftrightarrow C) \times (A \longleftrightarrow B)$	$A \longleftrightarrow C$	$((R \mathbin{⨾} (\ni)) \cap (S \mathbin{⨾} \pi_1^{-1})) \mathbin{⨾} \pi_2$

> **Note:** If ρ is a function and c is an member of the domain of ρ then the following equality holds: $(\_c)\,\rho \;=\; \rho\, c$. So we have the following equivalence:
>
> $$(a, b) \in (\_c) \;\Leftrightarrow\; (c, b) \in a$$
>
> If the relations $R : A \longleftrightarrow (B \longleftrightarrow C)$ and $S : A \longleftrightarrow B$ both have the element a in their domains, then the tuple $(a, (b, c))$ belongs to $(R \mathbin{⨾} (\ni))$ providing that (b, c) is a member of the set $R(a)$ and it belongs to $(S \mathbin{⨾} \pi_1^{-1})$ if b is $S(a)$. The tuple (a, c) belongs to the whole relation exactly when for some b the tuple $(a, (b, c))$ belongs to the first part. If R and S are functions, then promoted application is defined so that the following equality holds:
>
> $$(R \bullet S)(a) \;=\; (R\,a)(S\,a)$$

Promoted application is disjunctive in both arguments.

The apply-to-a function $(\_a)$ can be derived from promoted application as follows:

$$(\_a) \;=\; I \bullet a^\circ$$

□

5 Semantic universe

Notes on this section of the Z Standard

Section title: Semantic universe
Section editor: Randolph Johnson
Original text by: Stephen Brien, Jim Woodcock, Peter Lupton
Source file: semdom.tex
Most recent update: 12th September 1995
Formatted: 14th September 1995

5.1 Introduction

This section defines a semantic universe within which the meanings of Z specifications lie. The approach is based on the Zermelo-Fraenkel axiomatisation of sets, as indicated in the previous section, Section 4, *Semantic metalanguage*.

The syntax of Z defines a set of specifications. The semantics of Z defines a function from these specifications to meanings within the semantic universe. This universe 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. It should be noted that the meaning of a specification is further paramatrised by the assignment of sets to given set names in the specification.

5.2 Names and types

The first task in building the 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 abstract names is dependent on the specification in question, the members of each set not being distinguishable in the concrete syntax. Abstractly, we have that, for any particular Z specification, our set of names, *Name*, is comprised of schema names, variable names, and constant names:

$$SchemaName \cup Variable \cup Constant = Name$$

Representation names can have different abstract forms for different specifications; there is assumed to be an infinite supply of each.

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.

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

$$GivenSetName \subseteq Constant$$

Note: The names $\mathbb{Z}$ for the set of integers are members of the set of given set names. That is, $\{\mathbb{Z}\} \subseteq GivenSetName$.

Every type belongs to the 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, respectively.

Basic familiarity with elementary set theory leads one to view something of given type as an object, of power set type as a set, of Cartesian product type as a tuple, but what about something of schema type? It is a partial function from variable names to types; such a function is called a signature:

$$Signature \mathrel{\widehat{=}} Variable \mathrel{\twoheadrightarrow\!\!\!\!\!\!+} Type$$

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 represented by σ, we can construct the unique type which is represented by $\mathbb{P}\,\sigma$; every power set type is constructed in this way from a unique type. Thus, the power set type 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*:

$$\begin{array}{l}
givenT : GivenSetName \rightarrowtail\!\!\!\!\twoheadrightarrow Gtype \\
powerT : Type \rightarrowtail\!\!\!\!\twoheadrightarrow Ptype \\
cproductT : (Type^{+} \setminus Type^{1}) \rightarrowtail\!\!\!\!\twoheadrightarrow Ctype \\
schemaT : Signature \rightarrowtail\!\!\!\!\twoheadrightarrow Stype
\end{array}$$

Note: The signature parameter for the *schemaT* operator can be the empty signature.

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. In terminology from category theory, *Type* can be described as the initial algebra over the signature given by *givenT*, *powerT*, *cproductT*, *schemaT*. Using the notation for free types as defined in Z, we can sum this up by defining the set *Type* as follows:

$$\begin{array}{rl}
Type ::= & givenT\langle\!\langle GivenSetName\rangle\!\rangle \\
\mid & powerT\langle\!\langle Type\rangle\!\rangle \\
\mid & cproductT\langle\!\langle (Type^{+} \setminus Type^{1})\rangle\!\rangle \\
\mid & schemaT\langle\!\langle Signature\rangle\!\rangle
\end{array}$$

5.3 Values in Z

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 (ZF) 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 by defining the carrier function for given types and then constructing the function for other types from this.

> **Note:** In Z a type is represented by its carrier set.

A set W will be defined below to contain the values of all elements in Z. The carrier sets for each type in Z are subsets of W. The set $\mathcal{W}_0$ is the set comprising the carrier sets for each of the given types.

Definition 5.1 *For each specification there is a carrier function which maps the given types to elements of* $\mathcal{W}_0$*.*

$$\mathit{Carrier}_0 : \mathit{Gtype} \longrightarrow \mathcal{W}_0$$

> **Note:** It is always the case that $\mathit{Carrier}_0\,(\,\mathbb{Z}\,) \;=\; \{\,\ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots\}$.

> **Note:** The carrier sets (elements of $\mathcal{W}_0$) may be empty sets. This means that the types are not inhabited.

> **Note:** Suppose that γ is a given type; what is the carrier set of the power set type $\mathit{powerT}\,\gamma$? It is simply the set $\mathbb{P}(\mathit{Carrier}\,\gamma)$. In general, if γ in a different situation is a power set type of a given type T, we must calculate the carrier set by stripping off the power set constructor, calculating the carrier set of this underlying given type, and then forming the power set of the result. This is given by the expression
>
> $$(\mathit{powerT}^{-1} \fatsemi \mathit{Carrier}_0 \fatsemi (\mathbb{P}))\,\gamma$$
>
> Similarly, if γ is a Cartesian product of given types, then we should break it up into its constituent given types, determine their carrier sets, and then form their Cartesian product, so that we end up with a set of tuple values:
>
> $$(\mathit{cproductT}^{-1} \fatsemi \mathit{Carrier}_0^{\oplus} \fatsemi (\times))\,\gamma$$
>
> Finally, if γ is a schema type constructed from given types, then we should obtain the underlying signature; this yields a function from variable names to types, which we must turn into a function from variable 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:
>
> $$(\mathit{schemaT}^{-1} \fatsemi {}^{\exists}(I_{\mathit{Variable}} \;\otimes\; \mathit{Carrier}_0) \fatsemi \mathcal{X})\,\gamma$$
>
> The indexed product operator $\mathcal{X}$ is used to convert a function $\mathit{Variable} \ffun \mathbb{P}\,W$ to a set of functions $\mathbb{P}(\mathit{Variable} \ffun W)$.

In this discussion, we have assumed 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 5.2

$$\begin{aligned}
&\mathit{Carrier}_{i+1} \mathrel{\widehat{=}} \\
&\quad \mathit{Carrier}_i \\
&\quad \cup\, \mathit{powerT}^{-1} \fatsemi \mathit{Carrier}_i \fatsemi (\mathbb{P}) \\
&\quad \cup\, \mathit{cproductT}^{-1} \fatsemi \mathit{Carrier}_i^{\oplus} \fatsemi (\times) \\
&\quad \cup\, \mathit{schemaT}^{-1} \fatsemi {}^{\exists}\, (I_{\mathit{Variable}} \otimes \mathit{Carrier}_i) \fatsemi \mathcal{X}
\end{aligned}$$

> **Note:** The carrier set of the schema type constructed from the empty signature is the set containing the empty binding.

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

Definition 5.3 *The general carrier function mapping 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 $\mathcal{W}_0$ using the type constructors.

Definition 5.4 *The set W of all values is the set of all the elements in each of the carrier sets for the elements of Type:*

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

Definition 5.5 *A binding is a finite mapping from variables to values:*

$$\mathit{Binding} \mathrel{\widehat{=}} \mathit{Variable} \ffun W$$

The carrier function is a homomorphism between *Type* and W.

> **Note:** Thus, we have the equations for carrier
>
> $$\begin{aligned}
> &\mathit{Carrier}\,(\mathit{powerT}\,\gamma) = \mathbb{P}\,(\mathit{Carrier}\,\gamma) \\
> &\mathit{Carrier}\,(\mathit{cproductT}\,(\gamma_1, \ldots, \gamma_n)) = (\mathit{Carrier}\,\gamma_1) \times \ldots \times (\mathit{Carrier}\,\gamma_n) \\
> &\mathit{Carrier}\,(\mathit{schemaT}\,\gamma) = \mathcal{X}({}^{\exists}\,(I \otimes \mathit{Carrier})\;\gamma)
> \end{aligned}$$

5.4 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.

> **Editor's note:** The commuting diagram "The type system" has been temporarily omitted from this section.

Definition 5.6 *The set Elm is a set of type-value pairs:*

$$Elm : \mathbb{P}\,(Type \times W)$$

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

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

> **Note:** The set *Elm* of all compatible type-value pairs is also a relation between types and elements of their carrier sets. If the carrier set for a type is empty, then the type will not be in the domain of *Elm*.

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

Definition 5.7 *The type and value functions are projections from the tuples in Elm:*

$$t \mathrel{\widehat{=}} Elm \lhd \pi_1$$
$$v \mathrel{\widehat{=}} Elm \lhd \pi_2$$

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

Definition 5.8 *The set Pelm contains all elements which have power type:*

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

Definition 5.9 *The membership relation, ∃, for elements in Z is a relation between Pelm and Elm:*

$$\exists \;\; : Pelm \longleftrightarrow Elm$$

This relation is the product of the relation between a power type and its underlying type and the relation between a set and its members:

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

A Z specification 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 5.10 *A situation is a finite mapping from variables to elements:*

$$Situation \mathrel{\widehat{=}} Variable \mathrel{⇻} Elm$$

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 5.11 *The $T$and Vfunctions are defined as follows:*

$$\begin{array}{lcl} T & \mathrel{\widehat{=}} & {}^{\exists}(I_{Variable} \otimes t) \\ V & \mathrel{\widehat{=}} & {}^{\exists}(I_{Variable} \otimes v) \end{array}$$

The following commuting diagram, in which n is an arbitrary variable name, illustrates the relationship between types and values and their lifted forms as signatures and bindings:

$$\begin{tikzcd}
Signature \arrow[d, "(\_n)"'] & Situation \arrow[l, "T"'] \arrow[r, "V"] \arrow[d, "(\_n)"] & Binding \arrow[d, "(\_n)"] \\
Type & Elm \arrow[l, "t"] \arrow[r, "v"'] & W
\end{tikzcd}$$

5.5 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 $\varnothing[\mathbb{Z}]$, we know this has a different type from $\varnothing[\mathbb{P}\,\mathbb{Z}]$. 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.

5.5.1 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 elements is given a type. So each generic type is a function from fixed-length sequences of power types to a type. In order to simplify the definition of generic types we consider first the case of the generic type with n parameters and then extend to the general case. This type is a function from an n-tuple of power types to a type. This function must be total as all possible combinations of parameters must be given a type.

Definition 5.12 *For any natural number $n > 0$, the set of all generic types with n parameters is defined as follows:*

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

Since the number of parameters for a generic type is fixed, the general generic type is an example of one of the specific fixed length generic types. So the set of all generic types is the union of all the fixed length types.

Definition 5.13 *The set of all generic types is the union of all the sets of finite length generic types:*

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

> **Note:** This models a set far bigger than that which can be constructed using generic definitions in Z

5.5.2 Generic elements

As with generic types, for each generic element there is a fixed 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. Generic elements are defined in a similar way to generic types: by defining the specific n-length case and generalising.

For any natural number $n > 0$, the set of all generic elements with n parameters is a subset of the set of functions from n-tuples of set elements to elements:

$$Gen\_Elm_n : \mathbb{P}(Pelm^n \longrightarrow Elm)$$

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. This is an important restriction on the functions used to model generic elements. In order to define this property it is necessary to characterise the type part of a generic element.

Definition 5.14 *The function τ_{n} takes a function from n-tuples of elements to elements and returns a relation from n-tuples of type to type:*

$$\tau_{\mathrm{n}} : (Pelm^{\mathrm{n}} \longrightarrow Elm) \longrightarrow (Ptype^{\mathrm{n}} \longleftrightarrow Type)$$

$$\tau_{\mathrm{n}} \mathrel{\widehat{=}} {}^{\exists}(t^{\mathrm{n}} \otimes t)$$

The functions which are to be characterised as generic elements are those whose type part is a generic type, i.e. those whose type part is functional.

Definition 5.15 *The set of generic elements with n parameters are those functions whose type part is functional, i.e. contained in* Gen_Type_n*:*

$$Gen\_Elm_n \;\widehat{=}\; \mathrm{dom}\,(\tau_{\mathrm{n}} \rhd Gen\_Type)$$

In the same way as for generic types, the general generic element is an example of a specific one for some fixed number of parameters.

Definition 5.16 *The set of all generic elements is the union of all the sets of finite length generic elements:*

$$Gen\_Elm \;\widehat{=}\; Gen\_Elm_1 \cup Gen\_Elm_2 \cup \ldots$$

Definition 5.17 *The generic type function can be generalised as follows:*

$$\tau \;\widehat{=}\; \tau_1 \cup \tau_2 \cup \ldots$$

5.6 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. The meaning of a Z specification is a set of environments. This set contains those environments which map the names declared in the specification to a combination of values which correspond to the constraints within the specification.

Definition 5.18 *An environment is defined as a finite partial function from names to elements or generic elements:*

$$Env \;\widehat{=}\; Name \unicode{x21FB} (Elm \cup Gen\_Elm)$$

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 simply necessary to have an environment in which the types of all names are recorded.

Definition 5.19 *A type-environment is defined as a finite function from names to types or generic types:*

$$Tenv \;\widehat{=}\; Name \unicode{x21FB} (Type \cup Gen\_Type)$$

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

$$\Upsilon \,\widehat{=}\, {}^{\exists}(I_{Name} \;\otimes\; (t \cup \tau))$$

The following commuting diagram, in which n is an arbitrary name, illustrates the relationship between the environment and the type-environment:

$$\begin{tikzcd}
\mathit{Tenv} \arrow[d, "(\_n)"'] & \mathit{Env} \arrow[l, "\Upsilon"'] \arrow[d, "(\_n)"] \\
\mathit{Type} \cup \mathit{Gen\_Type} & \mathit{Elm} \cup \mathit{Gen\_Elm} \arrow[l, "t \cup \tau"]
\end{tikzcd}$$

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

5.7 Generic lifting

Editor's note: The following text is by Peter Lupton and replaces the paper on Generic lifting originally presented at the York meeting in April 1995.

Intuitively, a generic declaration is understood in terms of an environment enriched by assigning all possible meanings to the generic parameters. This intuition also provides a welcome contribution towards economy: evaluation of the expression used to declare the generic name re-uses existing expression evaluation. Of course, a user of the Z language has no need to think in terms of all possible meanings of the generic parameters, and will evaluate generic parameters lazily, on an as-used basis. But semantically, we need to represent the completed totality of such possible evaluations.

Generic lifting is the formal equivalent of the intuition outlined above. There are two version of generic lifting — one for the type environment and one for the meaning environment. We consider the meaning case first.

We are given a way of evaluating declarations, which amounts to a relation R:

$$R : \mathit{Env} \longleftrightarrow \mathit{situation}$$

By means of the semantic equations of a specification, we are able to relate each augmented environment with a collection of situations.

Our task is to re-use R defined above to define the generically lifted version of R, where R is to be lifted by treating $X_1, \ldots, X_n$ as generic variables:

$${}^{(X_1,\ldots,X_n)}(R)^{\mathcal{M}} \; : \; \mathit{Env} \longleftrightarrow \mathit{Env}$$

The process which takes us from R to $^{(X_1,\ldots,X_n)}\,(R)^{\mathcal{M}}$ can be defined in the following steps:

1. Starting with some environment ρ, augment it by enriching ρ with all possible meanings for $X_1, \ldots, X_n$, while simultaneously keeping track of the meanings used in that enrichment.
2. The relation R is then applied to the augmented environments, to produce situations. At this point, we have a relation which, given the meanings used to augment an environment and that environment, delivers situations according to R.
3. Our relation maps meanings to situations; in order for this to become an environment, we must switch the meanings and the names which appear as the domain of a situation. This gives an environment which is the lifted form of R.

The first structure needed is that of *indexed_situation*. An indexed_situation is a subset of:

$$constant^{+} \times (Pelm^{+} \times situation)$$

It is to be understood that, given an n-tuple of generic variables and an n-tuple of possible meanings of these variables, a situation is defined which maps each generic variable to its corresponding meaning:

$$indexed\_situation \;=\; {}^{\exists}\,(\langle \pi_1, \langle \pi_2\,, \otimes^{-1} \,⨾\, \{..\}\rangle\,\rangle)\ .\ (constant \times Ptype \,⨾\, \langle I,\, Pelm\rangle)^{\otimes}$$

The augment function. We want to show how indexed_situations can be used to define enrichments to environments. This is done by introducing the function $augment^{\mathcal{M}}$

$$augment^{\mathcal{M}} \;\in\; constant^{+} \longrightarrow (Env \longleftrightarrow (Pelm^{+} \times Env))$$

That is, given generic parameters, an environment and a generic argument, an environment is defined which is to be the original environment enriched by the corresponding situation.

$$augment^{\mathcal{M}} \;=\; {}^{\Lambda}\left({}^{\exists}\,(switch \,⨾\, \langle \pi_1, \pi_2 \,⨾\, \langle \pi_2 \,⨾\, \pi_1, \langle \pi_1, \pi_2 \,⨾\, \pi_2\rangle \,⨾\, (\oplus)\rangle\,\rangle)\ .\ Env \times indexed\_situation\right)$$

where *switch* is defined as follows:

$$switch = \langle \pi_2 \,⨾\, \pi_1, \langle \pi_1, \pi_2 \,⨾\, \pi_2\rangle\rangle$$

The function *augment* completes the first part of the three-step process described above. The second two steps are now defined.

$$^{(X_1,\ldots,X_n)}\,(R)^{\mathcal{M}} \;=\; {}^{\Lambda}\left(augment^{\mathcal{M}}(X_1, \ldots, X_n) \,⨾\, (I \otimes R)\right) \,⨾\, recurry$$

where *recurry* takes a curried function and interchanges the arguments:

$$recurry\ :\ A \longleftrightarrow (B \longleftrightarrow C) \;\longrightarrow\; B \longleftrightarrow (A \longleftrightarrow C)$$

The definition of generic lifting of type information is similar, except that it must be established that the generic expression is well typed for every possible assignment of power type to the generic parameters. The definition of generic lifting for types is given below, largely without comment.

We are given a way of assigning types to certain type environments:

$$R \;:\; \mathit{Tenv} \mathrel{\rightarrow\!\!\!\!\!\!+\;} \mathit{Signature}$$

The task is to re-use the function R to define a generically lifted version of R where $X_1, \ldots, X_n$ are to be treated as generic parameters:

$$^{(X_1, \ldots, X_n)}\,(R)^{\mathcal{T}} \;:\; \mathit{Tenv} \mathrel{\rightarrow\!\!\!\!\!\!+\;} \mathit{Tenv}$$

$$\mathit{indexed\_signature} \;=\; ^{\exists}\,(\langle \pi_1, \langle \pi_2\,,\, \otimes^{-1} \fatsemi \{..\}\rangle\rangle)\;.\;(\mathit{constant} \times \mathit{Ptype})^{\otimes}$$

$$\mathit{augment}^{\mathcal{T}} \;=\; ^{\Lambda}\Big(^{\exists}\,(\mathit{switch} \fatsemi \langle \pi_1, \pi_2 \fatsemi \langle \pi_2 \fatsemi \pi_1, \langle \pi_1, \pi_2 \fatsemi \pi_2\rangle \fatsemi (\oplus)\rangle\,\rangle)\;.\;\mathit{Tenv} \times \mathit{indexed\_signature}\Big)$$

$$^{(X_1, \ldots, X_n)}\,(R)^{\mathcal{T}} \;=\; ^{\Lambda}\Big(\mathit{augment}^{\mathcal{T}}(X_1, \ldots, X_n) \fatsemi (I \otimes R)\Big) \fatsemi \mathit{total} \fatsemi \mathit{recurry}$$

The above definition ensures that every possible assignment of power type to the generic parameters gives rise to a well-typed value. This is done by ensuring that the function concerned is total.

The function *total* is defined as follows:

$$\mathit{total}_n = I \rhd ((\mathit{Ptype}^n \longrightarrow (\mathbb{P}\,\mathit{signature} \setminus \{\phi\})) \fatsemi (\ni))$$

$$\mathit{total} = \mathit{total}_1 \cup \mathit{total}_2 \cup \ldots$$

□

6 Language description

Notes on this section of the Z Standard

Section title: Language description
Section editor: Randolph Johnson
Contributions by: . . . *(to be added)*
Source file: lang.tex
Most recent update: 14th September 1995
Formatted: 14th September 1995

6.1 Introduction

This section provides a general introduction to the following sections of the language definition, each of which defines a major syntactic category: *expression*, *predicate*, *schema*, *paragraph*, *specification*. Within each section there are subsections corresponding to the syntactic categories of the abstract syntax. These follow a consistent pattern, sub-divided as follows: *Abstract syntax*, *Concrete form*, *Sample representation and transformation*, *Type*, and *Value/Meaning*.

A *denotational* style of semantic description is used, as described for example in [27] 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, respectively, the superscripts $\mathcal{T}, \mathcal{V}, \mathcal{M}$ on the brackets.

Table 16: Brackets used for semantic functions

Bracket	Argument	Forms
$\llbracket \_ \rrbracket$	Expression	$\llbracket \_ \rrbracket^{\mathcal{T}}, \llbracket \_ \rrbracket^{\mathcal{V}}, \llbracket \_ \rrbracket^{\mathcal{M}}$
$\lBrace \_ \rBrace$	Predicate	$\lBrace \_ \rBrace^{\mathcal{T}}, \lBrace \_ \rBrace^{\mathcal{V}}, \lBrace \_ \rBrace^{\mathcal{M}}$
$\llparenthesis \_ \rrparenthesis$	Schema	$\llparenthesis \_ \rrparenthesis^{\mathcal{T}}, \llparenthesis \_ \rrparenthesis^{\mathcal{M}}$
$\lAngle \_ \rAngle$	Paragraph	$\lAngle \_ \rAngle^{\mathcal{T}}, \lAngle \_ \rAngle^{\mathcal{M}}$
$\lAngle \_ \rAngle$	Specification	$\lAngle \_ \rAngle^{\mathcal{T}}, \lAngle \_ \rAngle^{\mathcal{M}}$

The following meta-variables are used for expressing the *representation syntax*, i.e., the visual form in which it appears. Similar symbols, in bold font, are used for expressing the abstract syntax.

Table 17: Meta-variables used in the representation syntax

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

6.2 Abstract syntax

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

$$\mathsf{POWERSET} \ ::= \ \mathbf{Pow}\ \mathsf{EXP}$$

In some cases symbols such as $\boldsymbol{\theta}$ are used rather than key-words or other structures in the syntax to make reading of the abstract syntax easier. The abstract syntax is presented in Annex A.

6.3 Concrete form, representation and transformation

For each language element, there is an example of the concrete form showing a production or productions of the language element being defined, together with a table showing the relationship between the representation 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.

Transformations are presented in a denotational style. Superscripts on brackets denote the type of the argument.

Table 18: Transformation functions

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

The following example illustrates how a sample form from the concrete syntax is presented, together with a metalanguage version of the representation and its corresponding abstract form:

Concrete form

```
PSET Expression
```

Sample representation and transformation

Representation	Abstract
$\mathbb{P}\ s$	$\mathbf{Pow}[\![s]\!]^{\mathcal{E}}$

In this example the production for power set shows how a power set is represented as an expression prefixed with the power set symbol. The first column in the table gives an example of the representation form. In this case s is some expression for a set in representational form. The second column gives the abstract form of this 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:

$$[\![\mathbb{P}\ s]\!]^{\mathcal{E}} \quad = \quad \mathbf{Pow}\ [\![s]\!]^{\mathcal{E}}$$

The concrete syntax is presented in Annex B.

6.4 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:** Determining whether a given specification is well-typed is a decidable question. Similarly, the determination of the type of any term, within a given environment, is decidable. This is in contrast with evaluation – determining whether a term has a certain value is, in general, undecidable.

Table 19: Type functions of major forms

Name	Form	Sort
Expression Type	$[\![\boldsymbol{E}]\!]^{\mathcal{T}}$	$Tenv \rightarrow\!\!\!\!\!\!+ Type$
Predicate Type	$\{\!\{ \boldsymbol{P} \}\!\}^{\mathcal{T}}$	$\mathbb{P}\ Tenv$
Schema Type	$(\!(\boldsymbol{S})\!)^{\mathcal{T}}$	$Tenv \rightarrow\!\!\!\!\!\!+ Signature$
Paragraph Type	$\langle\!\langle \boldsymbol{Par} \rangle\!\rangle^{\mathcal{T}}$	$Tenv \rightarrow\!\!\!\!\!\!+ Tenv$

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

> **Type** The type of the power set **Pow** $\boldsymbol{s}$ is the power set type of the type of the set $\boldsymbol{s}$.
>
> $$[\![\mathbf{Pow}\,\boldsymbol{s}]\!]^{\mathcal{T}} \quad = \quad ([\![\boldsymbol{s}]\!]^{\mathcal{T}} \rhd Ptype) \fatsemi powerT$$
>
> **Note:** A power set **Pow** $\boldsymbol{s}$ is well typed only if $\boldsymbol{s}$ 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.

6.5 Meaning

The meanings of *expression, predicate, schema* and *paragraph* are given by the functions in the following table:

Table 20: Meaning functions of major forms

Name	Form	Sort
Expression Meaning	$[\![\, \boldsymbol{E} \,]\!]^{\mathcal{M}}$	$Env \pfun Elm$
Predicate Meaning	$\{\!\|\, \boldsymbol{P} \,\|\!\}^{\mathcal{M}}$	$\mathbb{P}\, Env$
Schema Meaning	$(\!\|\, \boldsymbol{S} \,\|\!)^{\mathcal{M}}$	$Env \longleftrightarrow Situation$
Schema Meaning	$(\!\|\, \boldsymbol{S} \,\|\!)^{\mathcal{MT}}$	$Env \longleftrightarrow Situation$
Paragraph Meaning	$\langle\!\|\, \boldsymbol{Par} \,\|\!\rangle^{\mathcal{M}}$	$Env \longleftrightarrow Env$

Editor's note: There is a need to explain that the definition of Env is paramatrised by the assignment of values to the given sets.

The meanings of expression, predicate, and schema are combined to provide a meaning for a paragraph. This meaning is a relation between environments. A specification is regarded as a sequence of paragraphs. The meaning of a specification then is defined as the image of the empty environment under the composite relation formed by composing the relations for all the paragraphs in the specification.

The following example illustrates the description of the meaning of a schema construction:

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

$$(\!|\, \langle \boldsymbol{D} \mid \boldsymbol{P} \rangle \,|\!)^{\mathcal{M}} \quad = \quad (\!|\, \boldsymbol{D} \,|\!)^{\mathcal{M}} \cap (\langle\!|\, \boldsymbol{D} \mid \boldsymbol{P} \,|\!\rangle^{\mathcal{M}} \mathbin{⨾} (\supseteq))$$

This is defined only in those environments in which the schema $\boldsymbol{D}$ is defined and when enriched by it result in the predicate $\boldsymbol{P}$ being well-typed.

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

The meanings of pairs of expressions, predicates, etc., can be compared. For example, two expressions e_1 and e_2 are said to be semantically equivalent, written $e_1 \equiv e_2$, when $[\![\, \boldsymbol{e}_1 \,]\!]^{\mathcal{M}} = [\![\, \boldsymbol{e}_2 \,]\!]^{\mathcal{M}}$.

6.6 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 and have the structure shown in the following table:

Table 21: Value functions of major forms

Name	Form	Sort
Expression Value	$[\![\boldsymbol{E}]\!]^{\mathcal{V}}$	$Env \mathrel{+\!\!\!\to} W$
Predicate Value	$\{\!\{ \boldsymbol{P} \}\!\}^{\mathcal{V}}$	$\mathbb{P}\, Env$

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

Value The value of the power set **Pow** $\boldsymbol{s}$ is the set of all the subsets of the value of $\boldsymbol{s}$:

$$[\![\mathbf{Pow}\, \boldsymbol{s}]\!]^{\mathcal{V}} \quad = \quad [\![\boldsymbol{s}]\!]^{\mathcal{V}} \mathbin{;} (\mathbb{P})$$

Note: A powerset **Pow** $\boldsymbol{s}$ is value-defined only if the expression $\boldsymbol{s}$ is value-defined.

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

6.7 Free variables

Ordinarily the definition of the free variables of an expression can be considered as a function on the names of identifiers appearing in the text of the expression and the variables bound by the declarations. In Z however, the 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 related not 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 function from environments to sets of names:

$$\phi(E) : Env \mathrel{+\!\!\!\to} \mathbb{P}\, Name$$

The set of names defined as the free variables of an expression in 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.

In Annex F, *The logical theory of Z*, there are equations giving the free variables for the various constructs of expressions, predicates, schemas and paragraphs. The free variable functions in the following table are used. Note that two different functions are needed to handle schemas. A schema may play different roles and the free variables it has when regarded as an expression may differ from its free variables as a predicate.

Table 22: Free variable functions

Function	Argument
ϕ	Expression
Φ	Predicate
ϕ	Schema
ϕ	Paragraph

The following example illustrates the definition of the free variables of a power set:

$$\phi(\mathbb{P}\ \boldsymbol{s}) \quad = \quad \phi(\boldsymbol{s})$$

6.8 Alphabet

The syntactic category of schema is used to introduce new names. These new names are called the *alphabet*. The alphabet is the set of the names in the domain of the signature function *Signature* as defined in Section 5.

In Annex F, *The logical theory of Z*, there are equations giving the alphabet for the various constructs of schemas and paragraphs. The alphabet function in the following table is used.

Table 23: Alphabet function

Function	Argument
α	Paragraph
α	Schema

The following example illustrates the definition of the alphabet of a simple schema:

$$\alpha(\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m : \boldsymbol{s}) \quad = \quad \{\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m\}$$

6.9 Substitution

In Annex F, *The logical theory of Z*, tables of semantic equivalences are given for substitutions in expressions, predicates, schema expressions and schema predicates. These tables indicate when one expression, etc. can be replaced by another without changing the meaning. Substitution is defined using a binding, which asssigns values to variable names. These new values are substituted for the variables in the expression.

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

Table 24: Extract from table of semantic equivalences

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

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

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

where the symbol $\equiv$ denotes semantic equivalence.

> **Note:** The following is an example of substitution:
>
> $$\langle\!\vert\, x := 5,\ y := \mathbb{N},\ a := 7 \,\vert\!\rangle \odot (x \in y \cap z) \quad \equiv \quad 5 \in \mathbb{N} \cap z$$
>
> Since the variable name a does not occur free in the expression, there is no substitution for it.

$\Box$

7 Expression

Notes on this section of the Z Standard

Section title: Expression
Section editor: (for this version, edited by JEN).
Contributions by: Stephen Brien, Randolph Johnson, Jon Hall, ... *(others to be added)*
Source file: exp.tex
Most recent update: 13th September 1995
Formatted: 14th September 1995

7.1 Introduction

Expression is a general form for defining values. In this section, 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.

> **Note:** Facilities for defining arithmetic and other operators for use in expressions are included in Annex D (Mathematical Toolkit), where they are defined in Z.

In the abstract syntax given below, the different kinds of Z expression are listed.

Abstract Syntax

EXP ::=	IDENT	Identifier
|	GENINST	Generic instantiation
|	NUMBERL	Number literal
|	SETEXTN	Set extension
|	SETCOMP	Set comprehension
|	POWERSET	Power set
|	TUPLE	Tuple
|	PRODUCT	Cartesian product
|	TUPLESELECTION	Tuple selection
|	BINDINGEXTN	Binding extension
|	THETAEXP	Theta expression
|	SCHEMAEXP	Schema expression
|	BINDSELECTION	Binding selection
|	FUNCTAPP	Application
|	DEFNDESCR	Definite description
|	IFTHENELSE	Conditional expression
|	EXPSUBSTITUTION	Substitution

Semantic brackets. In the following sections, the brackets $[\![$ and $]\!]$ are used to express properties of the environment for expressions. See Section 6, *Language Description*, for definitions of these brackets.

Type function. For any expression $\boldsymbol{E}$, its *type function* $[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{T}}$ is a recursively defined partial function from type-environments to types:

$$[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{T}} : \mathit{Tenv} \mathrel{+\!\!\!\rightarrow} \mathit{Type}$$

The expression $\boldsymbol{E}$ is *well-typed* in exactly those type-environments contained in $\mathrm{dom}\,[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{T}}$. The type of an expression in a type-environment is the result of applying its type function to that type-environment.

Value function. For any expression $\boldsymbol{E}$, its *value function* $[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{V}}$ is a partial function from environments to values:

$$[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{V}} : \mathit{Env} \mathrel{+\!\!\!\rightarrow} W$$

The expression $\boldsymbol{E}$ is *value-defined* in exactly those environments contained in $\mathrm{dom}\,[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{V}}$.

The value of an expression in an environment is the result of the application of its value function to that environment.

For some expressions the semantics is loosely defined. That is to say, a lower bound on the meaning is given. This provides the possibility of various interpretations, in circumstances where the semantics does not explicitly give a meaning.

> **Note:** An example is the definition of Application. For example, in function application, when the argument is outside the domain of the function, then no meaning is explicitly given. Different interpretations of Z can ascribe different meanings to an ill-formed function application.

Meaning function. The *meaning function* $[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{M}}$ is a function from environments to elements:

$$[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{M}} : \mathit{Env} \mathrel{+\!\!\!\rightarrow} \mathit{Elm}$$

The meaning of an expression in an environment is the pair consisting of its type and its value in that environment. The type of an expression in an environment is its type evaluated in the corresponding type-environment, which is a restriction of the environment.

The function $\Upsilon \mathbin{⨾} [\![\ \boldsymbol{E}\ ]\!]^{\mathcal{T}}$ corresponds to the type function for $\boldsymbol{E}$ in the full meaning environment, where Υ is the function that restricts an environment to its corresponding type-environment. Thus the meaning function is constructed as follows:

$$[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{M}} = \langle \Upsilon \mathbin{⨾} [\![\ \boldsymbol{E}\ ]\!]^{\mathcal{T}}\ ,\ [\![\ \boldsymbol{E}\ ]\!]^{\mathcal{V}} \rangle$$

The expression $\boldsymbol{E}$ is *well-defined* in exactly those environments contained in $\mathrm{dom}\,[\![\ \boldsymbol{E}\ ]\!]^{\mathcal{M}}$ which is equal to:

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

An expression is said to be well-typed in an environment if it is well-typed in the corresponding type environment. Thus, an expression is well-defined in those environments in which it is well-typed and value-defined.

A result of this definition is that the type part 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.

$$\vdash [\![\boldsymbol{E}]\!]^{\mathcal{M}} \mathbin{;} t \subseteq \Upsilon \mathbin{;} [\![\boldsymbol{E}]\!]^{\mathcal{T}}$$

> **Note:** This relationship is not an equality because it is possible to have well-typed expressions which are not value-defined.

7.2 Identifier

An identifier is a name used to refer to a variable or a constant. It denotes a value which depends on its environment.

Abstract syntax

$\mathsf{IDENT} ::= \mathsf{NAME}$

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

Concrete form

`x`

Sample representation and transformation

Representation	Abstract
n	$[\![n]\!]^{\mathcal{E}}$

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

$$[\![\, \boldsymbol{n} \,]\!]^{\mathcal{T}} \quad = \quad (\_\boldsymbol{n}) \rhd \mathit{Type}$$

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

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

$$[\![\, \boldsymbol{n} \,]\!]^{\mathcal{V}} \quad = \quad (\_\boldsymbol{n}) \fatsemi v$$

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

7.3 Generic instantiation

The generic instantiation $\boldsymbol{n}_{[s_1,\ldots,s_m]}$ $(m > 0)$, is the instantiation of the generic variable $\boldsymbol{n}$ with the list of set expressions $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m$. The number m must be equal to the number of formal parameters of $\boldsymbol{n}$. Each element of the instantiation list gives a value to the corresponding generic parameter of the generic variable.

If the list of generic parameters is omitted in the representation form, it is inferred from the typing information in the context of use; in this case, the values of 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 one or more expressions.

GENINST ::= NAME [EXP, EXP, . . .]

Concrete form

```
Name SQBRA ExpressionList1 SQKET
```

Editor's note: The concrete definitions of generic instantation will be updated using templates.

Sample representation and transformation Generic variables can be instantiated by providing a parameter list, or by infix or prefix means.

Representation	Abstract
$n_{[s_1,\ldots,s_m]}$	$\llbracket n \rrbracket^{\mathcal{E}} \, [\llbracket s_1 \rrbracket^{\mathcal{E}}, \ldots, \llbracket s_m \rrbracket^{\mathcal{E}}]$
$s_1 \psi s_2$	$\llbracket (\_\psi\_) \rrbracket^{\mathcal{E}} \, [\llbracket s_1 \rrbracket^{\mathcal{E}}, \llbracket s_2 \rrbracket^{\mathcal{E}}]$
ϕs	$\llbracket (\phi\_) \rrbracket^{\mathcal{E}} \, [\llbracket s \rrbracket^{\mathcal{E}}]$

> **Note:** The expression $s_1 \psi s_2$, where ψ is an infix generic symbol is the variable $(\_\psi\_)$ when instantiated with the parameter list $[s_1, s_2]$. When ϕ is a prefix generic symbol then ϕs is the variable declared as $(\phi\_)$ when instantiated with the parameter list $[s]$.

Type The type of a generic instantiation $\boldsymbol{n}_{[s_1,\ldots,s_m]}$ is obtained by applying to the types of the actual parameters $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m$ the function corresponding to the generic type of the variable name $\boldsymbol{n}$ in the type-environment:

$$\llbracket\ \boldsymbol{n}_{[s_1,\ldots,s_m]}\ \rrbracket^{\mathcal{T}} \quad = \quad (\_\boldsymbol{n}) \bullet \langle \llbracket\ \boldsymbol{s}_1\ \rrbracket^{\mathcal{T}}, \ldots, \llbracket\ \boldsymbol{s}_m\ \rrbracket^{\mathcal{T}} \rangle$$

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

Value The value of a generic instantiation $\boldsymbol{n}_{[s_1,\ldots s_m]}$ is obtained by applying the function corresponding to the generic meaning of the variable name $\boldsymbol{n}$ in the environment to the meanings of the actual parameters $\boldsymbol{s}_1, \ldots \boldsymbol{s}_m$ and then taking the value part:

$$[\![\ \boldsymbol{n}_{[s_1,\ldots,s_m]} \ ]\!]^{\mathcal{V}} \quad = \quad ((\_\boldsymbol{n}) \bullet \langle [\![\ \boldsymbol{s}_1 \ ]\!]^{\mathcal{M}}, \ldots, [\![\ \boldsymbol{s}_m \ ]\!]^{\mathcal{M}} \rangle) \ ⨾ \ v$$

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

7.4 Number literal

A number literal denotes an integer.

Abstract syntax

$$\mathsf{NUMBERL} ::= \mathsf{NUMBER}$$

Concrete form

`NUMBER`

Sample representation and transformation

Representation	Abstract
i	$\llparenthesis i \rrbracket^{\mathcal{E}}$

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

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

Note: A number literal is always well-typed.

Value The value of a number literal is the integer it denotes.

$$[\![\, \boldsymbol{i} \,]\!]^{\mathcal{V}} \quad = \quad \boldsymbol{i}^{\circ}$$

Note: A number literal is always value-defined.

7.5 Set extension

A non-empty set extension $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ is a set containing exactly those elements denoted by $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ $(m > 0)$.

Abstract syntax A set extension is constructed from a list of one or more expressions.

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

Concrete form

```
SETBRA ExpressionList SETKET
```

Sample representation and transformation

Representation	Abstract
$\{x_1, \ldots, x_m\}$	$\boldsymbol{\{}[\![(x_1)]\!]^{\mathcal{E}}, \ldots, [\![(x_m)]\!]^{\mathcal{E}}\boldsymbol{\}}$

Type The type of a set extension $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ is the power set type of the common type of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$.

$$[\![\ \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}\ ]\!]^{\mathcal{T}} \quad = \quad ([\![\ \boldsymbol{x}_1\ ]\!]^{\mathcal{T}} \cap \ldots \cap [\![\ \boldsymbol{x}_m\ ]\!]^{\mathcal{T}}) \mathbin{⨾} \mathit{powerT}$$

Note: A set extension $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ is well typed if and only if all of the expressions $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ are well-typed with the same type.

Note: If σ represents the common type of $x_1, \ldots, x_m$, then $\mathbb{P}\,\sigma$ represents the type of the set extension $\{x_1, \ldots, x_m\}$, $\mathbb{P}(\mathbb{Z} \times \sigma)$ represents the type of the sequence $\langle x_1, \ldots, x_m\rangle$ and $\mathbb{P}(\sigma \times \mathbb{Z})$ represents the type of the bag $[\![x_1, \ldots, x_m]\!]$.

Value The value of a set extension $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ is the set of the values of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$:

$$[\![\ \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}\ ]\!]^{\mathcal{V}} \quad = \quad \langle [\![\ \boldsymbol{x}_1\ ]\!]^{\mathcal{V}}, \ldots, [\![\ \boldsymbol{x}_m\ ]\!]^{\mathcal{V}}\rangle \mathbin{⨾} \{..\}$$

Note: A set extension $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ is value-defined if and only if all of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ are value-defined.

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

7.6 Set comprehension

The set comprehension $\boldsymbol{\{TE \bullet x\}}$ is the set that contains exactly those elements denoted by the expression $\boldsymbol{x}$ when evaluated in each enrichment of the current environment by the text-or-expression $\boldsymbol{TE}$.

Abstract syntax A set comprehension is constructed from a text-or-expression and an expression.

$$\mathsf{SETCOMP} ::= \{\mathsf{SCHEMA} \bullet \mathsf{EXP}\}$$

Concrete form

```
 SETBRA TextOrExpression DOT Expression SETKET
SETBRA Text SETKET
LAMBDA TextOrExpression DOT Expression
```

Sample representation and transformation There are two ways of constructing a set by comprehension: a simple set (for which the expression part is optional) and a lambda expression.

Representation	Abstract
$\{TE \bullet x\}$	$\{\llbracket TE \rrbracket^{\mathcal{TE}} \bullet \llbracket x \rrbracket^{\mathcal{E}}\}$
$\{TE\}$	$\{\llbracket TE \rrbracket^{\mathcal{TE}} \bullet \llbracket (TE)^{\chi} \rrbracket^{\mathcal{E}}\}$
$\lambda\, TE \bullet x$	$\{\llbracket TE \rrbracket^{\mathcal{TE}} \bullet (\llbracket (TE)^{\chi} \rrbracket^{\mathcal{E}}, \llbracket x \rrbracket^{\mathcal{E}})\}$

.

> **Note:** If the expression part of the set comprehension is omitted then the default is the characteristic tuple of the text-or-expression

> **Note:** A lambda expression denotes a function. The parameter is the characteristic tuple of the text-or-expression . The domain is defined by the text-or-expression . The value of the function for a given parameter is defined by the value of the Expression for the value of that parameter.

Editor's note: A definition of χ is needed.

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

$$[\![\ \{\boldsymbol{TE} \bullet \boldsymbol{x}\}\ ]\!]^{\mathcal{T}} \quad = \quad \{\!|\ \boldsymbol{TE}\ |\!\}^{\mathcal{T}} \mathbin{;} [\![\ \boldsymbol{x}\ ]\!]^{\mathcal{T}} \mathbin{;} powerT$$

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

Value The value of a set comprehension $\{\boldsymbol{TE} \bullet \boldsymbol{x}\}$, is the set of the values denoted by the expression $\boldsymbol{x}$ in each of the enrichments of the environment by the text-or-expression $\boldsymbol{TE}$:

$$[\![\ \{\boldsymbol{TE} \bullet \boldsymbol{x}\}\ ]\!]^{\mathcal{V}} \quad = \quad {}^{\Lambda}\,(\{\!|\ \boldsymbol{TE}\ |\!\}^{\mathcal{M}} \mathbin{;} [\![\ \boldsymbol{x}\ ]\!]^{\mathcal{V}})$$

Note: A set comprehension is always value-defined.

7.7 Power set

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

Abstract syntax A power set is constructed from an expression.

POWERSET ::= **Pow** EXP

Concrete form

```
PSET Expression
```

Sample representation and transformation

Representation	Abstract
$\mathbb{P}\ s$	$\mathbf{Pow}[\![s]\!]^{\mathcal{E}}$

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

$$[\![\ \mathbf{Pow}\, s\ ]\!]^{\mathcal{T}} \;=\; ([\![\ s\ ]\!]^{\mathcal{T}} \rhd \mathit{Ptype}) \fatsemi \mathit{powerT}$$

Note: A power set **Pow**s is well-typed if and only if s has power set type.

Note: If **Pow**σ represents the type of the set s, then **Pow Pow** σ represents the type of **Pow** s, a set of sets.

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

$$[\![\ \mathbf{Pow}\, s\ ]\!]^{\mathcal{V}} \;=\; [\![\ s\ ]\!]^{\mathcal{V}} \fatsemi (\mathbb{P})$$

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

7.8 Tuple

A tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$ $(m > 1)$ is an ordered collection of the elements $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$. The elements $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ are not required to have the same type.

> **Note:** The tuples (a, b, c) and $((a, b), c)$ are distinct: the first contains three components a, b, c whereas the second has components (a, b) and c.

> **Note:** 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.

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

Concrete form

```
BRA Expression COMMA ExpressionList1 KET
```

Sample representation and transformation

Representation	Abstract
$(x_1, \ldots, x_m)$	$([\![x_1]\!]^{\mathcal{E}}, \ldots, [\![x_m]\!]^{\mathcal{E}})$

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

$$[\![\ (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)\ ]\!]^{\mathcal{T}} \quad = \quad \langle [\![\ \boldsymbol{x}_1\ ]\!]^{\mathcal{T}}, \ldots, [\![\ \boldsymbol{x}_m\ ]\!]^{\mathcal{T}} \rangle \fatsemi \mathit{cproductT}$$

> **Note:** A tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)(m > 1)$ is well-typed if and only if all of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ are well-typed.

Value The value of a tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)(m > 1)$ is the tuple formed from the values of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$:

$$[\![\ (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)\ ]\!]^{\mathcal{V}} \quad = \quad \langle [\![\ \boldsymbol{x}_1\ ]\!]^{\mathcal{V}}, \ldots, [\![\ \boldsymbol{x}_m\ ]\!]^{\mathcal{V}} \rangle$$

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

7.9 Cartesian product

The expression $s_1 \times \ldots \times s_m$ $(m > 1)$ is the Cartesian product of the sets $s_1, \ldots, s_m$.

The sets $s_1, \ldots, s_m$ are not required to have the same type.

> **Note:** As with tuples, the Cartesian products $a \times b \times c$ and $(a \times b) \times c$ are distinct.

Abstract syntax A Cartesian product is constructed from two or more expressions.

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

Concrete form

```
Expression CROSS Expression { CROSS Expression }
```

Sample representation and transformation

Representation	Abstract
$s_1 \times \ldots \times s_m$	$[\![s_1]\!]^{\mathcal{E}} \times \ldots \times [\![s_m]\!]^{\mathcal{E}}$

Type The type of a Cartesian product $s_1 \times \ldots \times s_m (m > 1)$ is the power set type of the Cartesian product type of the list of the underlying types of the sets $s_1, \ldots, s_m$.

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

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

Value The value of a Cartesian product $s_1 \times \ldots \times s_m (m > 1)$ is the Cartesian product of the values of the sets $(s_1, \ldots, s_m)$:

$$[\![\ s_1 \times \ldots \times s_m\ ]\!]^{\mathcal{V}} \quad = \quad \langle [\![\ s_1\ ]\!]^{\mathcal{V}}, \ldots, [\![\ s_m\ ]\!]^{\mathcal{V}} \rangle \mathbin{;} (\times)$$

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

> **Note:** If $x_i \in s_i$ for $1 \leq i \leq m$, then the tuple $(x_1, \ldots, x_m)$ is an element of $s_1 \times \ldots \times s_m$.

7.10 Tuple selection

The tuple selection $\boldsymbol{t.i}$ is the ith element in the tuple $\boldsymbol{t}$.

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

$\mathsf{TUPLESELECTION} ::= \mathsf{EXP}\ .\ \mathsf{NUMBER}$

Note: The syntactic category $\mathsf{NUMBERL}$ is used to ensure well-typedness of selection.

Concrete form

```
Expression SELECT NUMBER
```

Sample representation and transformation

Representation	Abstract
$t.i$	$[\![t]\!]^{\mathcal{E}} . [\![i]\!]^{\mathcal{E}}$

Type The type of a tuple selection $\boldsymbol{t.i}$ is the type of the ith element of the tuple $\boldsymbol{t}$.

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

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

Value The value of a tuple selection $\boldsymbol{t.i}$ is the value of the ith element of the tuple $\boldsymbol{t}$.

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

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

7.11 Binding extension

A binding extension $\langle\!| \boldsymbol{n}_1 := \boldsymbol{x}_1, \ldots, \boldsymbol{n}_m := \boldsymbol{x}_m |\!\rangle$ $(m > 0)$ is the binding that maps the names $\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m$ to the values of the expressions $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ respectively.

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

$$\mathsf{BINDINGEXTN} ::= \langle\!| \mathsf{NAME} := \mathsf{EXP},\ \mathsf{NAME} := \mathsf{EXP},\ \ldots |\!\rangle$$

Concrete form

```
SQBRA BindList SQKET
```

Sample representation and transformation

Representation	Abstract
$\langle\!| n_1 := x_1, \ldots, n_m := x_m |\!\rangle$	$\langle\!| \llbracket \boldsymbol{n}_1 \rrbracket^{\mathcal{E}} := \llbracket x_1 \rrbracket^{\mathcal{E}}, \ldots, \llbracket \boldsymbol{n}_m \rrbracket^{\mathcal{E}} := \llbracket x_m \rrbracket^{\mathcal{E}} |\!\rangle$

Type The type of a binding extension $\langle\!| \boldsymbol{n}_1 := \boldsymbol{x}_1, \ldots, \boldsymbol{n}_m := \boldsymbol{x}_m |\!\rangle$ $(m > 0)$ is the schema type of the signature constructed from the mapping of the names $\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m$ to the types of the expressions $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$.

$$\llbracket\ \langle\!| \boldsymbol{n}_1 := \boldsymbol{x}_1, \ldots, \boldsymbol{n}_m := \boldsymbol{x}_m |\!\rangle\ \rrbracket^{\mathcal{T}} \quad = \quad \langle \langle \boldsymbol{n}_1{}^{\circ}, \llbracket\ \boldsymbol{x}_1\ \rrbracket^{\mathcal{T}} \rangle, \ldots, \langle \boldsymbol{n}_m{}^{\circ}, \llbracket\ \boldsymbol{x}_m\ \rrbracket^{\mathcal{T}} \rangle \rangle \fatsemi \{..\} \fatsemi \mathit{schemaT}$$

> **Note:** A binding extension $\langle\!| \boldsymbol{n}_1 := \boldsymbol{x}_1, \ldots, \boldsymbol{n}_m := \boldsymbol{x}_m |\!\rangle$ is well-typed if and only if the expressions $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ are all well-typed, and the names are distinct.

Value The value of a binding extension $\langle\!| \boldsymbol{n}_1 := \boldsymbol{x}_1, \ldots, \boldsymbol{n}_m := \boldsymbol{x}_m |\!\rangle$ $(m > 0)$ is the binding constructed from the mapping of the names $\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m$ to the values of the expressions $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$.

$$\llbracket\ \langle\!| \boldsymbol{n}_1 := \boldsymbol{x}_1, \ldots, \boldsymbol{n}_m := \boldsymbol{x}_m |\!\rangle\ \rrbracket^{\mathcal{V}} \quad = \quad \langle \langle \boldsymbol{n}_1^{\circ}, \llbracket\ \boldsymbol{x}_1\ \rrbracket^{\mathcal{V}} \rangle, \ldots, \langle \boldsymbol{n}_m^{\circ}, \llbracket\ \boldsymbol{x}_m\ \rrbracket^{\mathcal{V}} \rangle \rangle \fatsemi \{..\}$$

> **Note:** A binding extension $\langle\!| \boldsymbol{n}_1 := \boldsymbol{x}_1, \ldots, \boldsymbol{n}_m := \boldsymbol{x}_m |\!\rangle$ is value-defined if and only if the expressions $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ are all value-defined.

7.12 Theta expression

The theta expression $\boldsymbol{\theta}\,\boldsymbol{S}$ is the binding whose type is constructed from the signature of $\boldsymbol{S}$ 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 $\boldsymbol{\theta}\,\boldsymbol{S}^{\,q}$ is the binding whose type is constructed from the signature of $\boldsymbol{S}$ 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}.

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

$$\mathsf{THETAEXP} ::= \boldsymbol{\theta}\,\mathsf{SCHEMA\ DECOR}$$

Note: The schema may itself be decorated. Thus the following are permitted: $\theta\,S^{\,q}$ and $\theta\,(S^{q})^{\,q}$.

Note: Only non-generic schemas may be used in theta expressions.

Concrete form

```
THETA Expression
```

Sample representation and transformation

Representation	Abstract
$\theta\ \ S^{\,q}$	$\boldsymbol{\theta}\llbracket S\rrbracket^{\mathcal{S}}\llbracket {}^{q}\rrbracket^{\mathcal{E}}$
$\theta\ \ S$	$\boldsymbol{\theta}\,\llbracket S\rrbracket^{\mathcal{S}}$

Type The type of $\boldsymbol{\theta}\,\boldsymbol{S}$ ($\boldsymbol{\theta}\,\boldsymbol{S}^{\,q}$) is the schema type constructed from the signature of $\boldsymbol{S}$ whose components (when decorated by ^{q}) have the same non-generic type as the corresponding variable in the type-environment:

$$\begin{aligned}\llbracket\, \boldsymbol{\theta S}\, \rrbracket^{\mathcal{T}} &= (\llparenthesis\, \boldsymbol{S}\, \rrparenthesis^{\mathcal{T}} \cap \supseteq) \fatsemi schemaT\\ \llbracket\, \boldsymbol{\theta S}^{q}\, \rrbracket^{\mathcal{T}} &= \Big(\llparenthesis\, \boldsymbol{S}\, \rrparenthesis^{\mathcal{T}} \cap (\sqsupseteq \fatsemi {}^{\exists}(\llparenthesis\, \boldsymbol{q}\, \rrparenthesis^{\mathcal{N}} \otimes I))\Big) \fatsemi schemaT\end{aligned}$$

Note: A theta expression is well-typed if and only if each of the decorated versions of the names of the signature of the schema is assigned a non-generic type in the type-environment which is the same as the type of that name in the signature.

Note: The type of a theta expression $\boldsymbol{\theta}\,\boldsymbol{S}\,^{q}$ is *not* the type taken from $\boldsymbol{S}$ 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 type-environment; thus $\boldsymbol{\theta}\,\boldsymbol{S}\,^{r}$ and $\boldsymbol{\theta}\,\boldsymbol{S}\,^{q}$ are of the same type even if ^{r} and ^{q} are different decorations.

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

$$
\begin{aligned}
[\![\ \boldsymbol{\theta S}\ ]\!]^{\mathcal{V}} &= \Upsilon \mathbin{⨾} (\!|\ \boldsymbol{S}\ |\!)^{\mathcal{T}} \mathbin{⨾} \mathit{schemaT} \mathbin{⨾} \mathit{Elm} \cap \ \supseteq \mathbin{⨾}\ V \\
[\![\ \boldsymbol{\theta S}^{q}\ ]\!]^{\mathcal{V}} &= \Upsilon \mathbin{⨾} (\!|\ \boldsymbol{S}\ |\!)^{\mathcal{T}} \mathbin{⨾} \mathit{schemaT} \mathbin{⨾} \mathit{Elm} \cap \ \supseteq \mathbin{⨾}\ ^{\exists}(\langle\!|\ \boldsymbol{q}\ |\!\rangle^{\mathcal{N}} \ \otimes \ v)
\end{aligned}
$$

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.

7.13 Schema expression

A schema expression $\boldsymbol{S}$ is the set of bindings defined by the schema $\boldsymbol{S}$.

Abstract syntax A schema expression is constructed from a schema.

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

Concrete form

```
TextOrExpression
```

Sample representation and transformation

Representation	Abstract
S	$[(S)]^{S}$

Type The type of a schema expression $\boldsymbol{S}$ is the power set type of the schema type constructed from the signature of the schema $\boldsymbol{S}$:

$$[\![\ \boldsymbol{S}\ ]\!]^{\mathcal{T}} \;=\; (\!|\ \boldsymbol{S}\ |\!)^{\mathcal{T}} \fatsemi \mathit{schemaT} \fatsemi \mathit{powerT}$$

Note: A schema expression $\boldsymbol{S}$ is well-typed if and only if the schema $\boldsymbol{S}$ is well-typed.

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

Value The value of a schema expression $\boldsymbol{S}$ is the set of bindings defined by the schema $\boldsymbol{S}$:

$$[\![\ \boldsymbol{S}\ ]\!]^{\mathcal{V}} \;=\; {}^{\Lambda}\,((\!|\ \boldsymbol{S}\ |\!)^{\mathcal{M}} \fatsemi V)$$

Note: A schema expression $\boldsymbol{S}$ is always value-defined.

7.14 Binding selection

The binding selection $\boldsymbol{b.n}$ is the element to which the name $\boldsymbol{n}$ is mapped in the binding $\boldsymbol{b}$.

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

$$\mathsf{BINDSELECTION} ::= \mathsf{EXP}\ .\ \mathsf{NAME}$$

Concrete form

```
Expression SELECT Name
```

Sample representation and transformation

Representation	Abstract
$b.n$	$[\![b]\!]^{\mathcal{E}} \,.\, [\![n]\!]^{\mathcal{E}}$

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

$$[\![\ \boldsymbol{b}.\ \boldsymbol{n}\ ]\!]^{\mathcal{T}} \quad = \quad [\![\ \boldsymbol{b}\ ]\!]^{\mathcal{T}} \mathbin{⨾} schemaT^{-1} \mathbin{⨾} (\_\boldsymbol{n})$$

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

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

$$[\![\ \boldsymbol{b}.\ \boldsymbol{n}\ ]\!]^{\mathcal{V}} \quad = \quad [\![\ \boldsymbol{b}\ ]\!]^{\mathcal{V}} \mathbin{⨾} (\_\boldsymbol{n})$$

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

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

7.15 Application

The application $\boldsymbol{f}\ \boldsymbol{x}$ is the result of applying $\boldsymbol{f}$ to the argument $\boldsymbol{x}$.

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

> **Editor's note:** Additional text is needed in the description of Application, to take account of templates.

Abstract syntax

FUNCTAPP ::= EXP (EXP)

Concrete form

```
Expression Expression
```

Sample representation and transformation

Representation	Abstract
$f\ x$	$[\![f]\!]^{\mathcal{E}}\ [\![x]\!]^{\mathcal{E}}$

Type In the expression $\boldsymbol{f}\ \boldsymbol{x}$ the type of $\boldsymbol{f}$ must be the power set type of the Cartesian product type of a pair of types, and the type of the argument $\boldsymbol{x}$ must be the first type in this pair; the type of $\boldsymbol{f}\ \boldsymbol{x}$ is the second type in the pair.

$$[\![\ \boldsymbol{f}\ \boldsymbol{x}\ ]\!]^{\mathcal{T}} \quad = \quad \left([\![\ \boldsymbol{f}\ ]\!]^{\mathcal{T}} \mathbin{⨾} powerT^{-1} \mathbin{⨾} cproductT^{-1} \mathbin{⨾} \{-\}\right) \bullet [\![\ \boldsymbol{x}\ ]\!]^{\mathcal{T}}$$

Note: The application $\boldsymbol{f}\ \boldsymbol{x}$ is well-typed only if the type of $\boldsymbol{f}$ is a power set type of a pair of types with the first type in the pair the same as the type of $\boldsymbol{x}$.

Note: If we evaluate the type of $\boldsymbol{f}$, 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 $\boldsymbol{x}$, then we can simply use the type of $\boldsymbol{f}$ as a function to look up the type of the result corresponding to $\boldsymbol{x}$. We say the the type of $\boldsymbol{f}$ is essentially a set of pairs, because we must 'undo' the type constructors.

Value The value of an application $\boldsymbol{f}\ \boldsymbol{x}$ is given by applying the value of $\boldsymbol{f}$ to the value of the argument $\boldsymbol{x}$:

$$[\![\, \boldsymbol{f}\, \boldsymbol{x} \,]\!]^{\mathcal{V}} \ \supseteq\ {}^{\Lambda}([\![\, \boldsymbol{f} \,]\!]^{\mathcal{V}} \bullet [\![\, \boldsymbol{x} \,]\!]^{\mathcal{V}}) \mathbin{⨾} \{-\}^{-1}$$

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

Note: A relation is a set of pairs; the first element of each pair represents an argument, and the second the result for that argument. For the application $\boldsymbol{f}\,\boldsymbol{x}$ to be defined, $\boldsymbol{f}$ must be functional at $\boldsymbol{x}$. Providing that $\boldsymbol{x}$ 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:

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

The promoted application $[\![\, \boldsymbol{f} \,]\!]^{\mathcal{V}} \bullet [\![\, \boldsymbol{x} \,]\!]^{\mathcal{V}}$ provides a satisfactory meaning when the application is value-defined. It is necessary to decide what to do with $\boldsymbol{f}\,\boldsymbol{x}$ when $\boldsymbol{f}$ is not functional at $\boldsymbol{x}$. This arises if there are several pairs in the value of $\boldsymbol{f}$, each having the same first element equal to the value of $\boldsymbol{x}$ or if there is none. The definition provided does not prescribe a value for a relation applied outside its domain or where it is non-functional.

7.16 Definite description

The definite description $\boldsymbol{\mu TE \bullet x}$ is the element denoted by $\boldsymbol{x}$ in the unique enrichment of the environment by the text-or-expression $\boldsymbol{TE}$.

Abstract syntax A definite description is constructed from a text-or-expression and an expression.

$$\mathsf{DEFNDESCR} ::= \mu\,\mathsf{SCHEMA} \bullet \mathsf{EXP}$$

Concrete form

```
BRA MU TextOrExpression KET
```

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

Representation	Abstract
$\mu\ TE \bullet x$	$\boldsymbol{\mu}\llbracket TE \rrbracket^{\mathcal{TE}} \bullet \llbracket x \rrbracket^{\mathcal{E}}$
$\mu\ TE$	$\boldsymbol{\mu}\,\llbracket TE \rrbracket^{\mathcal{TE}} \bullet \llbracket (TE)^{\chi} \rrbracket^{\mathcal{E}}$

Note: If the expression part of the definite description is omitted then the default is the characteristic tuple of the text-or-expression .

Type The type of the term $\boldsymbol{\mu TE \bullet x}$ is the type of $\boldsymbol{x}$ in the type-environment enriched by $\boldsymbol{TE}$:

$$\llbracket\ \boldsymbol{\mu TE \bullet x}\ \rrbracket^{\mathcal{T}} \quad = \quad \llparenthesis\ \boldsymbol{TE}\ \rrparenthesis^{\mathcal{T}} \fatsemi \llbracket\ \boldsymbol{x}\ \rrbracket^{\mathcal{T}}$$

Note: The expression $\boldsymbol{\mu TE \bullet x}$ is well-typed if and only if $\boldsymbol{TE}$ is well-typed, and $\boldsymbol{x}$ is well-typed in the type-environment enriched by $\boldsymbol{TE}$.

Value The value of a definite description $\boldsymbol{\mu TE \bullet x}$ is the value of $\boldsymbol{x}$ in the unique enrichment of the environment by $\boldsymbol{TE}$:

$$\llbracket\ \boldsymbol{\mu TE \bullet x}\ \rrbracket^{\mathcal{V}} \quad \supseteq \quad {}^{\Lambda}(\llparenthesis\ \boldsymbol{TE}\ \rrparenthesis^{\mathcal{M}}) \fatsemi \{-\}^{-1} \fatsemi \llbracket\ \boldsymbol{x}\ \rrbracket^{\mathcal{V}}$$

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

Note: This definition is not specific about the value of an improper definite description. If there is no unique enrichment of the environment then the value is not prescribed; hence the use of $\supseteq$ in the definition.

7.17 Conditional expression

The conditional expression **if** $\boldsymbol{P}$ **then** $\boldsymbol{x}$ **else** $\boldsymbol{y}$ denotes an expression which is equal to $\boldsymbol{x}$ if the predicate $\boldsymbol{P}$ is true, otherwise it is equal to the expression $\boldsymbol{y}$.

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

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

Concrete form

```
IF Predicate THEN Expression ELSE Expression
```

Sample representation and transformation

Representation	Abstract
if P then x else y	$\mathbf{if}\,[\![P]\!]^{\mathcal{P}}\,\mathbf{then}\,[\![x]\!]^{\mathcal{E}}\,\mathbf{else}\,[\![y]\!]^{\mathcal{E}}$

Type The type of the conditional expression **if** $\boldsymbol{P}$ **then** $\boldsymbol{x}$ **else** $\boldsymbol{y}$ is the common type of the expressions $\boldsymbol{x}$ and $\boldsymbol{y}$ when the predicate $\boldsymbol{P}$ is well-typed:

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

Note: The expression **if** $\boldsymbol{P}$ **then** $\boldsymbol{x}$ **else** $\boldsymbol{y}$ is well-typed if and only if the predicate $\boldsymbol{P}$ is well-typed and the expressions $\boldsymbol{x}$ and $\boldsymbol{y}$ are both well-typed with the same type.

Value The value of the conditional expression **if** $\boldsymbol{P}$ **then** $\boldsymbol{x}$ **else** $\boldsymbol{y}$ is the value of the expression $\boldsymbol{x}$ when the predicate $\boldsymbol{P}$ is true, otherwise it is the value of the expression $\boldsymbol{y}$:

$$[\![\ \mathbf{if}\ \boldsymbol{P}\ \mathbf{then}\ \boldsymbol{x}\ \mathbf{else}\ \boldsymbol{y}\ ]\!]^{\mathcal{V}} \quad = \quad (\{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{M}} \lhd [\![\ \boldsymbol{x}\ ]\!]^{\mathcal{V}}) \cup (\{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{M}} \mathbin{⩤} [\![\ \boldsymbol{y}\ ]\!]^{\mathcal{V}})$$

Note: The expression **if** $\boldsymbol{P}$ **then** $\boldsymbol{x}$ **else** $\boldsymbol{y}$ is value-defined if and only if the predicate $\boldsymbol{P}$ is true and the expression $\boldsymbol{x}$ is value-defined or the predicate $\neg\boldsymbol{P}$ is true and the expression $\boldsymbol{y}$ is value-defined.

7.18 Substitution

The expression $\boldsymbol{b} \odot \boldsymbol{x}$ denotes an expression equal to $\boldsymbol{x}$ in the environment enriched by the binding $\boldsymbol{b}$.

Abstract syntax An expression substitution is constructed from a binding and an expression.

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

Concrete form

```
Expression SUBST Expression
```

Editor's note: The concrete form of Substitution is under review. A form of **let** statement is being considered.

Sample representation and transformation

Representation	Abstract
$b \odot x$	$[\![b]\!]^{\mathcal{E}} \odot [\![x]\!]^{\mathcal{E}}$

Type The type of the substitution $\boldsymbol{b} \odot \boldsymbol{x}$ is the type of the expression $\boldsymbol{x}$ in the type-environment enriched by the binding $\boldsymbol{b}$.

$$[\![\, \boldsymbol{b} \odot \boldsymbol{x} \,]\!]^{\mathcal{T}} \quad = \quad \langle I, [\![\, \boldsymbol{b} \,]\!]^{\mathcal{T}} \mathbin{;} schemaT^{-1} \rangle \mathbin{;} (\oplus) \mathbin{;} [\![\, \boldsymbol{x} \,]\!]^{\mathcal{T}}$$

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

Value The value of the substitution $\boldsymbol{b} \odot \boldsymbol{x}$ is the value of the expression $\boldsymbol{x}$ in the environment enriched by the binding $\boldsymbol{b}$.

$$[\![\, \boldsymbol{b} \odot \boldsymbol{x} \,]\!]^{\mathcal{V}} \quad = \quad \langle I, [\![\, \boldsymbol{b} \,]\!]^{\mathcal{M}} \mathbin{;} \diamond \rangle \mathbin{;} (\oplus) \mathbin{;} [\![\, \boldsymbol{x} \,]\!]^{\mathcal{V}}$$

Note: The substitution $\boldsymbol{b} \odot \boldsymbol{x}$ is value-defined if and only if $\boldsymbol{b}$ is value-defined and the expression $\boldsymbol{x}$ is value-defined in the environment enriched by the binding $\boldsymbol{b}$.

Editor's note: Revised versions of the following, previously included as part of the Expression section, are now in Annex F, *The logical theory of Z*:

Free variables
Substitution.

□

8 Predicate

Notes on this section of the Z Standard

Section title: Predicate
Section editor: (this version edited by JEN)
Original text by: Stephen Brien
Source file: pred.tex
Most recent update: 13th September 1995
Formatted: 14th September 1995

8.1 Introduction

Predicate is a general form for expressing properties of an environment, which may be expressed as truth-valued relationships between values of variables in the environment, or between other predicates. The literals TRUTH and FALSEHOOD are also defined.

The meaning of a predicate is described in two parts. The first part gives rules for determining whether it is well-typed or not. The second part determines whether the predicate is *ZF-true* in the environment. A predicate is *ZF-true* in an environment if the values of sub-expressions in the predicate are such that the predicate is true in that environment, without considering whether this predicate is well-typed.

In the abstract syntax listed below, the different kinds of predicate are listed.

Abstract syntax

PRED ::= EQUALITY	Equality
| MEMBERSHIP	Set membership
| TRUTH	Truth literal
| FALSEHOOD	False literal
| NEGATION	Negation
| DISJUNCTION	Disjunction
| CONJUNCTION	Conjunction
| IMPLICATION	Implication
| EQUIVALENCE	Equivalence
| UNIVERSALQUANT	Universal quantification
| EXISTSQUANT	Existential quantification
| UNIQUEQUANT	Unique existential quantification
| SPRED	Schema predicate
| PREDSUBSTITUTION	Predicate substitution

Semantic brackets. In the following sections, the brackets $\{\![$ and $]\!\}$ are used to express properties of the environment for predicates. See Section 6, *Language Description*, for definitions of these brackets.

Type function. At the level of the abstract syntax, it is known whether a given construct is a predicate, and the remaining concern is therefore whether the construct is well-typed. Hence the type function of a predicate is represented as the set of type-environments in which the predicate is well-typed.

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

Note: A predicate that is not well-typed in any environment has a type function that evaluates to the empty set.

Value function. The value function maps a predicate to the set of environments in which it is ZF-true.

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

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

Meaning function. The environments in which a predicate is true are exactly those environments in which the predicate is well-typed and is ZF-true.

$$\{\![\ \mathsf{PRED}\ ]\!\}^{\mathcal{M}} : \mathbb{P}\, Env$$

$$\{\![\ \mathsf{PRED}\ ]\!\}^{\mathcal{M}} \mathrel{\widehat{=}} {}^{\exists}\left(\Upsilon^{-1}\right) \{\![\ \mathsf{PRED}\ ]\!\}^{\mathcal{T}} \ \cap\ \{\![\ \mathsf{PRED}\ ]\!\}^{\mathcal{V}}$$

Note: As indicated in the note above, the predicate $\neg(x \in x)$ is ZF-true but not well-typed, hence it is not true in any environment. The meaning of the predicate is the empty set: $\{\![\ \boldsymbol{x} \in \boldsymbol{x}\ ]\!\}^{\mathcal{M}} = \varnothing$.

8.2 Equality

An equality relation between expressions is true if and only if the expressions have the same types and values.

Abstract syntax An equality is constructed from two expressions.

$$\mathsf{EQUALITY} ::= \mathsf{EXP} \ = \ \mathsf{EXP}$$

Concrete form

Note: The following form is derived from `Relation` and `InfixRel`

```
Expression EQUALS Expression
```

Sample representation and transformation

Representation	Abstract
$x = y$	$[\![x]\!]^{\mathcal{E}}=[\![y]\!]^{\mathcal{E}}$

Type An equality $\boldsymbol{x} = \boldsymbol{y}$ is well-typed in those environments in which the expressions $\boldsymbol{x}$ and $\boldsymbol{y}$ have the same type.

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

Value An equality $\boldsymbol{x} = \boldsymbol{y}$ is ZF-true in those environments in which the expressions $\boldsymbol{x}$ and $\boldsymbol{y}$ have the same values.

$$\{\![\, \boldsymbol{x} = \boldsymbol{y} \,]\!\}^{\mathcal{V}} \ = \ \mathrm{dom}([\![\, \boldsymbol{x} \,]\!]^{\mathcal{V}} \cap [\![\, \boldsymbol{y} \,]\!]^{\mathcal{V}})$$

8.3 Membership

The predicate $\boldsymbol{x} \in \boldsymbol{y}$ is true if and only if the expression $\boldsymbol{x}$ is a member of the set denoted by the expression $\boldsymbol{y}$.

Abstract syntax A membership predicate is constructed from two expressions.

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

Concrete form

Note: The following form is derived from `Relation` and `InfixRel`

```
Expression MEMBER Expression
```

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

Representation	Abstract
$x \in y$	$[\![x]\!]^{\mathcal{E}} \in [\![y]\!]^{\mathcal{E}}$

Type A predicate $\boldsymbol{x} \in \boldsymbol{y}$ is well-typed if and only if the type of the expression $\boldsymbol{y}$ is the power set type of that of the expression $\boldsymbol{x}$.

$$\{\![\, \boldsymbol{x} \in \boldsymbol{y} \,]\!\}^{\mathcal{T}} \quad = \quad \mathrm{dom}([\![\, \boldsymbol{x} \,]\!]^{\mathcal{T}} \mathbin{;} powerT \cap [\![\, \boldsymbol{y} \,]\!]^{\mathcal{T}})$$

Value A predicate $\boldsymbol{x} \in \boldsymbol{y}$ is ZF-true in exactly those environments in which the value of the expression $\boldsymbol{x}$ is a member of the value of the expression $\boldsymbol{y}$.

$$\{\![\, \boldsymbol{x} \in \boldsymbol{y} \,]\!\}^{\mathcal{V}} \quad = \quad \mathrm{dom}([\![\, \boldsymbol{x} \,]\!]^{\mathcal{V}} \cap [\![\, \boldsymbol{y} \,]\!]^{\mathcal{V}} \mathbin{;} (\ni))$$

8.4 Truth literal

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

Abstract syntax

$$\mathsf{TRUTH} ::= \mathbf{true}$$

Concrete form

```
TRUE
```

Sample representation and transformation

Representation	Abstract
true	**true**

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

$$\{\!\![\ \boldsymbol{true}\ ]\!\!\}^{\mathcal{T}} \quad = \quad Tenv$$

Value The truth literal **true** is ZF-true in all environments.

$$\{\!\![\ \boldsymbol{true}\ ]\!\!\}^{\mathcal{V}} \quad = \quad Env$$

8.5 False literal

The false literal **false** represents the predicate that is never true.

Abstract syntax

$\mathsf{FALSEHOOD} ::= \textbf{false}$

Concrete form

`FALSE`

Sample representation and transformation

Representation	Abstract
false	**false**

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

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

Value The false literal **false** is not ZF-true in any environment.

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

8.6 Negation

The negation $\neg \boldsymbol{P}$ is true if and only if the predicate $\boldsymbol{P}$ is not.

Abstract syntax A negation is constructed from a predicate.

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

Concrete form

```
NOT Predicate
```

Sample representation and transformation

Representation	Abstract
$\neg P$	$\neg [\![P]\!]^{\mathcal{P}}$

Type The negation $\neg \boldsymbol{P}$ is well-typed exactly when the predicate $\boldsymbol{P}$ is well-typed.

$$\{\!\!\{ \neg\ \boldsymbol{P} \}\!\!\}^{\mathcal{T}} \ = \ \{\!\!\{ \boldsymbol{P} \}\!\!\}^{\mathcal{T}}$$

Value The negation $\neg \boldsymbol{P}$ is ZF-true in those environments in which the predicate $\boldsymbol{P}$ is not ZF-true.

$$\{\!\!\{ \neg\ \boldsymbol{P} \}\!\!\}^{\mathcal{V}} \ = \ \mathit{Env} \setminus \{\!\!\{ \boldsymbol{P} \}\!\!\}^{\mathcal{V}}$$

8.7 Disjunction

The disjunction $\boldsymbol{P} \lor \boldsymbol{Q}$ is true if and only if at least one of the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ is true.

Abstract syntax A disjunction is constructed from two predicates.

$$\mathsf{DISJUNCTION} ::= \mathsf{PRED} \lor \mathsf{PRED}$$

Concrete form

```
Predicate OR Predicate
```

Sample representation and transformation

Representation	Abstract
$P \lor Q$	$[\![P]\!]^{\mathcal{P}} \lor [\![Q]\!]^{\mathcal{P}}$

Type The disjunction $\boldsymbol{P} \lor \boldsymbol{Q}$ is well-typed in exactly those type-environments in which both of the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are well-typed.

$$\{\![\, \boldsymbol{P} \lor \boldsymbol{Q} \,]\!\}^{\mathcal{T}} \;=\; \{\![\, \boldsymbol{P} \,]\!\}^{\mathcal{T}} \cap \{\![\, \boldsymbol{Q} \,]\!\}^{\mathcal{T}}$$

Value The disjunction $\boldsymbol{P} \lor \boldsymbol{Q}$ is ZF-true in exactly those environments in which one or both of the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are ZF-true.

$$\{\![\, \boldsymbol{P} \lor \boldsymbol{Q} \,]\!\}^{\mathcal{V}} \;=\; \{\![\, \boldsymbol{P} \,]\!\}^{\mathcal{V}} \cup \{\![\, \boldsymbol{Q} \,]\!\}^{\mathcal{V}}$$

8.8 Conjunction

The conjunction $\boldsymbol{P} \wedge \boldsymbol{Q}$ is true if both of the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are true.

Abstract syntax A conjunction is constructed from two predicates.

$$\mathsf{CONJUNCTION} ::= \mathsf{PRED} \wedge \mathsf{PRED}$$

Concrete form

```
Predicate AND Predicate
```

Sample representation and transformation There are three ways of constructing a conjunction: by a simple conjunction, by a compound relation, and by separating two or more predicates.

Representation	Abstract
$P \wedge Q$	$[\![P]\!]^{\mathcal{P}} \wedge [\![Q]\!]^{\mathcal{P}}$
$x_1\ \rho_1\ x_2\ \rho_2\ \ldots \rho_{n-1}\ x_n$	$[\![x_1\ \rho_1\ x_2]\!]^{\mathcal{P}} \wedge [\![x_2\ \rho_2\ \ldots \rho_{n-1}\ x_n]\!]^{\mathcal{P}}$
$P_1\ \mathsf{Sep}\ P_2 \ldots P_n$	$[\![P_1]\!]^{\mathcal{P}} \wedge [\![P_2]\!]^{\mathcal{P}} \ldots [\![P_n]\!]^{\mathcal{P}}$

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

Type The conjunction $\boldsymbol{P} \wedge \boldsymbol{Q}$ is well-typed in exactly those type-environments in which both of the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are well-typed.

$$\{\![\ \boldsymbol{P} \wedge \boldsymbol{Q}\ ]\!\}^{\mathcal{T}} \ = \ \{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{T}} \cap \{\![\ \boldsymbol{Q}\ ]\!\}^{\mathcal{T}}$$

Value The conjunction $\boldsymbol{P} \wedge \boldsymbol{Q}$ is ZF-true in exactly those environments in which both of the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are ZF-true.

$$\{\![\ \boldsymbol{P} \wedge \boldsymbol{Q}\ ]\!\}^{\mathcal{V}} \ = \ \{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{V}} \cap \{\![\ \boldsymbol{Q}\ ]\!\}^{\mathcal{V}}$$

8.9 Implication

The implication $\boldsymbol{P} \Rightarrow \boldsymbol{Q}$ is true if and only if the predicate $\boldsymbol{P}$ is false or the predicate $\boldsymbol{Q}$ is true.

Abstract syntax An implication is constructed from two predicates.

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

Concrete form

```
Predicate IMPLIES Predicate
```

Sample representation and transformation

Representation	Abstract
$P \Rightarrow Q$	$[\![P]\!]^{\mathcal{P}} \Rightarrow [\![Q]\!]^{\mathcal{P}}$

Type The implication $\boldsymbol{P} \Rightarrow \boldsymbol{Q}$ is well-typed in exactly those type-environments in which both the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are well-typed.

$$\{\![\, \boldsymbol{P} \Rightarrow \boldsymbol{Q} \,]\!\}^{\mathcal{T}} = \{\![\, \boldsymbol{P} \,]\!\}^{\mathcal{T}} \cap \{\![\, \boldsymbol{Q} \,]\!\}^{\mathcal{T}}$$

Value The implication $\boldsymbol{P} \Rightarrow \boldsymbol{Q}$ is true in exactly those environments in which the predicate $\boldsymbol{P}$ is not ZF-true or the predicate $\boldsymbol{Q}$ is ZF-true.

$$\{\![\, \boldsymbol{P} \Rightarrow \boldsymbol{Q} \,]\!\}^{\mathcal{V}} = (Env \setminus \{\![\, \boldsymbol{P} \,]\!\}^{\mathcal{V}}) \cup \{\![\, \boldsymbol{Q} \,]\!\}^{\mathcal{V}}$$

8.10 Equivalence

An equivalence $\boldsymbol{P} \Leftrightarrow \boldsymbol{Q}$ is true if and only if both predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are true or neither is true.

Abstract syntax An equivalence is constructed from two predicates.

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

Concrete form

```
Predicate IFF Predicate
```

Sample representation and transformation

Representation	Abstract
$P \Leftrightarrow Q$	$[\![P]\!]^{\mathcal{P}} \Leftrightarrow [\![Q]\!]^{\mathcal{P}}$

Type The equivalence $\boldsymbol{P} \Leftrightarrow \boldsymbol{Q}$ is well-typed in exactly those type-environments in which both the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are well-typed.

$$\{\![\ \boldsymbol{P} \Leftrightarrow \boldsymbol{Q}\ ]\!\}^{\mathcal{T}} = \{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{T}} \cap \{\![\ \boldsymbol{Q}\ ]\!\}^{\mathcal{T}}$$

Value The equivalence $\boldsymbol{P} \Leftrightarrow \boldsymbol{Q}$ is ZF-true in exactly those environments in which the predicates $\boldsymbol{P}$ and $\boldsymbol{Q}$ are both ZF-true or neither are ZF-true.

$$\{\![\ \boldsymbol{P} \Leftrightarrow \boldsymbol{Q}\ ]\!\}^{\mathcal{V}} = \{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{V}} \otimes \{\![\ \boldsymbol{Q}\ ]\!\}^{\mathcal{V}}$$

8.11 Universal quantification

The universally quantified predicate $\forall \boldsymbol{TE} \bullet \boldsymbol{P}$ is true if the predicate $\boldsymbol{P}$ is true for all possible combinations of values of the components of the text-or-expression $\boldsymbol{TE}$.

Abstract syntax A universal quantification is constructed from text-or-expression and a predicate.

$$\mathsf{UNIVERSALQUANT} \ ::= \ \forall \mathsf{SCHEMA} \bullet \mathsf{PRED}$$

Concrete form

```
FORALL TextOrExpression DOT Predicate
```

Sample representation and transformation

Representation	Abstract
$\forall TE \bullet P$	$\forall [\![(TE)]\!]^{\mathcal{TE}} \bullet [\![(P)]\!]^{\mathcal{P}}$

Type A universal quantification $\forall \boldsymbol{TE} \bullet \boldsymbol{P}$ is well-typed in a type-environment if and only if the predicate $\boldsymbol{P}$ is well-typed in that type-environment enriched by the text-or-expression $\boldsymbol{TE}$.

$$\{\![\ \forall \boldsymbol{TE} \bullet \boldsymbol{P} \ ]\!\}^{\mathcal{T}} \ = \ \mathrm{dom}(\langle\!| \ \boldsymbol{TE} \ |\!\rangle^{\mathcal{T}} \rhd \{\![\ \boldsymbol{P} \ ]\!\}^{\mathcal{T}})$$

Value A universal quantification $\forall \boldsymbol{TE} \bullet \boldsymbol{P}$ is ZF-true in an environment if and only if the predicate $\boldsymbol{P}$ is ZF-true in all enrichments of that environment by the text-or-expression $\boldsymbol{TE}$.

$$\{\![\ \forall \boldsymbol{TE} \bullet \boldsymbol{P} \ ]\!\}^{\mathcal{V}} \ = \ {}^{\forall}(\langle\!| \ \boldsymbol{TE} \ |\!\rangle^{\mathcal{M}}) \, \{\![\ \boldsymbol{P} \ ]\!\}^{\mathcal{V}}$$

8.12 Existential quantification

The existentially quantified predicate $\exists \boldsymbol{TE} \bullet \boldsymbol{P}$ is true if the predicate $\boldsymbol{P}$ is true for at least one possible combination of values of the components of the text-or-expression $\boldsymbol{TE}$.

Abstract syntax An existential quantification is constructed from text-or-expression and a predicate.

$$\mathsf{EXISTSQUANT} ::= \exists\, \mathsf{SCHEMA} \bullet \mathsf{PRED}$$

Concrete form

```
EXISTS TextOrExpression DOT Predicate
```

Sample representation and transformation

Representation	Abstract
$\exists\, TE \bullet P$	$\exists\, [\![TE]\!]^{\mathcal{TE}} \bullet [\![P]\!]^{\mathcal{P}}$

Type An existential quantification $\exists \boldsymbol{TE} \bullet \boldsymbol{P}$ is well-typed in a type-environment if and only if the predicate $\boldsymbol{P}$ is well-typed in that type-environment enriched by the text-or-expression $\boldsymbol{TE}$.

$$\{\!\!\{\, \exists \boldsymbol{TE} \bullet \boldsymbol{P} \,\}\!\!\}^{\mathcal{T}} \;=\; \mathrm{dom}(\langle\!|\, \boldsymbol{TE} \,|\!\rangle^{\mathcal{T}} \rhd \{\!\!\{\, \boldsymbol{P} \,\}\!\!\}^{\mathcal{T}})$$

Value An existential quantification $\exists \boldsymbol{TE} \bullet \boldsymbol{P}$ is ZF-true in an environment if and only if the predicate $\boldsymbol{P}$ is ZF-true in at least one enrichment of that environment by the text-or-expression $\boldsymbol{TE}$.

$$\{\!\!\{\, \exists \boldsymbol{TE} \bullet \boldsymbol{P} \,\}\!\!\}^{\mathcal{V}} \;=\; \mathrm{dom}(\langle\!|\, \boldsymbol{TE} \,|\!\rangle^{\mathcal{M}} \rhd \{\!\!\{\, \boldsymbol{P} \,\}\!\!\}^{\mathcal{V}})$$

8.13 Unique existential quantification

The unique existentially quantified predicate $\exists_1 \boldsymbol{TE} \bullet \boldsymbol{P}$ is true if the predicate $\boldsymbol{P}$ is true for exactly one possible combination of values of the components of the text-or-expression $\boldsymbol{TE}$.

Abstract syntax A unique existential quantification is constructed from a text-or-expression and a predicate.

$$\mathsf{UNIQUEQUANT} ::= \exists_1 \mathsf{SCHEMA} \bullet \mathsf{PRED}$$

Concrete form

```
EXISTS1 TextOrExpression DOT Predicate
```

Sample representation and transformation

Representation	Abstract
$\exists_1 TE \bullet P$	$\exists_1 [\![TE]\!]^{\mathcal{TE}} \bullet [\![P]\!]^{\mathcal{P}}$

Type A unique existential quantification $\exists_1 \boldsymbol{TE} \bullet \boldsymbol{P}$ is well-typed in a type-environment if and only if the predicate $\boldsymbol{P}$ is well-typed in that type-environment enriched by the text-or-expression $\boldsymbol{TE}$.

$$\{\![\exists_1 \boldsymbol{TE} \bullet \boldsymbol{P}]\!\}^{\mathcal{T}} = \mathrm{dom}(\langle\!| \boldsymbol{TE} |\!\rangle^{\mathcal{T}} \rhd \{\![\boldsymbol{P}]\!\}^{\mathcal{T}})$$

Value A unique existential quantification $\exists_1 \boldsymbol{TE} \bullet \boldsymbol{P}$ is ZF-true in an environment if and only if the predicate $\boldsymbol{P}$ is ZF-true in exactly one enrichment of that environment by the text-or-expression $\boldsymbol{TE}$.

$$\{\![\exists_1 \boldsymbol{TE} \bullet \boldsymbol{P}]\!\}^{\mathcal{V}} = \mathrm{dom}(^{\Lambda} (\langle\!| \boldsymbol{TE} |\!\rangle^{\mathcal{M}} \rhd \{\![\boldsymbol{P}]\!\}^{\mathcal{V}}) \mathbin{⨾} \{-\}^{-1})$$

8.14 Schema predicate

A schema predicate $\boldsymbol{S}$ is true if and only if the values of the components of the schema $\boldsymbol{S}$ are contained in the environment and their values satisfy the property of the schema.

Abstract syntax A schema predicate is constructed from a schema.

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

Concrete form

```
??  -- to be defined
```

Sample representation and transformation

Type A schema predicate $\boldsymbol{S}$ is well-typed in a type-environment if and only if the schema $\boldsymbol{S}$ is well-typed and the signature of $\boldsymbol{S}$ is contained in the environment.

$$\{\![\, \boldsymbol{S} \,]\!\}^{\mathcal{T}} \;=\; \mathrm{dom}((\![\, \boldsymbol{S} \,]\!)^{\mathcal{T}} \cap \supseteq)$$

Value A schema predicate $\boldsymbol{S}$ is ZF-true in an environment if and only if the environment contains a situation of the schema $\boldsymbol{S}$.

$$\{\![\, \boldsymbol{S} \,]\!\}^{\mathcal{V}} \;=\; \mathrm{dom}((\![\, \boldsymbol{S} \,]\!)^{\mathcal{M}} \cap \supseteq)$$

8.15 Substitution

The predicate $\boldsymbol{b} \odot \boldsymbol{P}$ is true if and only if the predicate $\boldsymbol{P}$ is true in the environment enriched by the binding $\boldsymbol{b}$.

Abstract syntax A substitution instance is constructed from an expression and a predicate.

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

Concrete form

```
Expression SUBST Predicate
```

Sample representation and transformation

Representation	Abstract
$b \odot P$	$[\![b]\!]^{\mathcal{E}} \odot [\![P]\!]^{\mathcal{P}}$

Type A predicate $\boldsymbol{b} \odot \boldsymbol{P}$ is well-typed in an type-environment if and only if $\boldsymbol{b}$ is well-typed with a schema type and the predicate $\boldsymbol{P}$ is well typed in the environment enriched by the binding.

$$\{\![\ \boldsymbol{b} \odot \boldsymbol{P}\ ]\!\}^{\mathcal{T}} \quad = \quad \mathrm{dom}(\langle I,\ [\![\ \boldsymbol{b}\ ]\!]^{\mathcal{T}} \mathbin{;} \mathit{schemaT}^{-1} \rangle \mathbin{;} (\oplus) \rhd \{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{T}})$$

Value The predicate $\boldsymbol{b} \odot \boldsymbol{P}$ is ZF-true in exactly those environments in which the binding $\boldsymbol{b}$ is value-defined and, when enriched by $\boldsymbol{b}$ make the predicate $\boldsymbol{P}$ ZF-true.

$$\{\![\ \boldsymbol{b} \odot \boldsymbol{P}\ ]\!\}^{\mathcal{V}} \quad = \quad \mathrm{dom}(\langle I,\ [\![\ \boldsymbol{b}\ ]\!]^{\mathcal{M}} \mathbin{;} \diamond \rangle \mathbin{;} (\oplus) \rhd \{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{V}})$$

Editor's note: Revised versions of the following, previously included as part of the Predicate section, are now in Annex F, *The logical theory of Z*:

Free variables
Substitution.

□

9 Schema

Notes on this section of the Z Standard

Section title: Schema
Source file: sch.tex
Section editor: (this version edited by JEN)
Original text by: Stephen Brien
Most recent update: 13th September 1995
Formatted: 14th September 1995

9.1 Introduction

A schema is an expression whose value is a set of bindings.

A schema can be used in the following ways:

as a declaration

as a predicate

as an operand of certain operators which construct schemas from other schemas.

When making schemas, the problem is not so much whether it is well defined (although a schema may fail to be defined). The problem is more to record the possible meanings of the declared names. The definition is built up in two stages. The *type* function defines the *signature* of a schema. The *meaning* relation relates the environment to those possible *situations* defined by the schema.

A schema can be also used to introduce new variables to the environment. A type and meaning enrichment function is given for this purpose.

Abstract syntax

```
SCHEMA ::= SDECL
         | SDES
         | SGENDES
         | SCONSTRUCTION
         | SNEGATION
         | SDISJUNCTION
         | SCONJUNCTION
         | SIMPLICATION
         | SEQUIVALENCE
         | SPROJECTION
         | SHIDING
         | SUNIVQUANT
         | SEXISTSQUANT
         | SUNIQUEQUANT
         | SRENAMING
         | EXPSCHEMA
```

Type function. For any schema $\boldsymbol{S}$ its *type* function is a recursively defined partial function from type-environments to signatures which record the types of the elements denoted by the variables introduced:

$$\llbracket\ \boldsymbol{S}\ \rrbracket^{\mathcal{T}} \ :\ Tenv \rightarrowtail\!\!\!\!\!\!+\ Signature$$

The schema $\boldsymbol{S}$ is *well-typed* in exactly those type-environments contained in dom $\llbracket\ \boldsymbol{S}\ \rrbracket^{\mathcal{T}}$. The signature of a schema in a type-environment is the result of applying its type function to that type-environment.

For any schema $\boldsymbol{S}$ its *type enrichment function* is a partial function from a type-environment to a new one in which the names of the constituent schema are known:

$$\llparenthesis\ \boldsymbol{S}\ \rrparenthesis^{\mathcal{T}} \ :\ Tenv \rightarrowtail\!\!\!\!\!\!+\ Tenv$$

$$\llparenthesis\ \boldsymbol{S}\ \rrparenthesis^{\mathcal{T}} \ =\ \langle I,\ \llbracket\ \boldsymbol{S}\ \rrbracket^{\mathcal{T}} \rangle \fatsemi (\oplus)$$

Meaning relation. A schema introduces names to the environment which can assume certain values. These values are not fixed. We can consider the meaning of a schema 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 schema as a relation between environments and situations. For any schema $\boldsymbol{S}$, its *meaning relation* is a relation from environments to situations:

$$\llbracket\ \boldsymbol{S}\ \rrbracket^{\mathcal{M}} \ :\ Env \longleftrightarrow Situation$$

The meaning of a schema in an environment is any one of the situations related to that environment by the meaning function. The meaning of a schema is partial because some schemas may fail – for example $n : s$ where s is undefined, or if s is an empty set. A schema $\boldsymbol{S}$ is *well-defined* in exactly those environments contained in $\mathrm{dom} \llbracket\, \boldsymbol{S} \,\rrbracket^{\mathcal{M}}$.

The meaning enrichment is represented as a relation between environments, for the same reason as the meaning of a schema as represented by a relation.

$$\langle\!|\, \boldsymbol{S} \,|\!\rangle^{\mathcal{M}} \;:\; Env \longleftrightarrow Env$$

$$\langle\!|\, \boldsymbol{S} \,|\!\rangle^{\mathcal{M}} \;=\; \langle I, \llbracket\, \boldsymbol{S} \,\rrbracket^{\mathcal{M}} \rangle \fatsemi (\oplus)$$

9.2 Schema designator

A schema designator is a schema name used to refer to schema. It may also contain a list of generic parameters 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.

$$\mathsf{SDES} ::= \mathsf{WORD}$$

Concrete form

```
Name SQBRA ExpressionList1 SQKET
```

Sample representation and transformation

Representation	Abstract
S	$\boldsymbol{S}$

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

$$\llbracket\, \boldsymbol{S} \,\rrbracket^{\mathcal{T}} \;=\; (I \bullet S^{\circ}) \mathbin{⨾} powerT^{-1} \mathbin{⨾} schemaT^{-1}$$

> **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 meaning of the reference in the environment.

$$\llbracket\, \boldsymbol{S} \,\rrbracket^{\mathcal{M}} \;=\; (I \bullet S^{\circ}) \mathbin{⨾} (\ni)$$

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

9.3 Generic schema designator

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

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

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

Concrete form

```
Name SQBRA ExpressionList1 SQKET
```

Sample representation and transformation

Representation	Abstract
$S_{[x_1,\ldots,x_n]}$	$\boldsymbol{S}[[\![x_1]\!]^{\mathcal{E}}, \ldots, [\![x_n]\!]^{\mathcal{E}}]$

Type

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

Meaning

$$(\!|\; \boldsymbol{S}[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n] \;|\!)^{\mathcal{M}} \quad = \quad ((I \bullet S^{\circ}) \bullet \langle x_1, \ldots, x_n \rangle) \fatsemi (\ni)$$

Note: Generically defined schemas must be instantiated.

9.4 Simple schema

A simple schema $\boldsymbol{n}_1, \ldots \boldsymbol{n}_m : \boldsymbol{s}$ introduces variables named $\boldsymbol{n}_1, \ldots \boldsymbol{n}_m$ whose values are drawn from the set $\boldsymbol{s}$.

Abstract syntax A simple schema is constructed from a list of names and an expression.

$$\mathsf{SDECL} ::= \mathsf{NAME},\ \mathsf{NAME}, \ldots\ :\ \mathsf{EXP}$$

Concrete form

```
NameList1 COLON Expression
```

Sample representation and transformation

Representation	Abstract
$n_1, n_2, \ldots, n_m : s$	$\boldsymbol{n}_1, \boldsymbol{n}_2, \ldots, \boldsymbol{n}_k :\ \llparenthesis s \rrparenthesis^{\mathcal{E}}$

Type The type of the simple schema $\boldsymbol{n}_1, \ldots \boldsymbol{n}_m : \boldsymbol{s}$ is the signature constructed from the names $\boldsymbol{n}_1, \ldots \boldsymbol{n}_m$ and the underlying type of the set expression $\boldsymbol{s}$.

$$\llparenthesis\ \boldsymbol{n}_1, \ldots, \boldsymbol{n}_m : \boldsymbol{s}\ \rrparenthesis^{\mathcal{T}} \quad = \quad \llbracket\ \boldsymbol{s}\ \rrbracket^{\mathcal{T}} \fatsemi \langle\langle \boldsymbol{n}_1{}^{\circ},\ powerT^{-1}\rangle, \ldots, \langle \boldsymbol{n}_m^{\circ}, powerT^{-1}\rangle\rangle \fatsemi \{..\}$$

> **Note:** The simple schema $\boldsymbol{n}_1, \ldots \boldsymbol{n}_m : \boldsymbol{s}$ is well-typed if and only if the expression $\boldsymbol{s}$ has power set type.

Meaning The meaning of the simple schema $\boldsymbol{n}_1, \ldots \boldsymbol{n}_m : \boldsymbol{s}$ is a relation from the environment to those situations which associate each of the names $\boldsymbol{n}_1, \ldots \boldsymbol{n}_m$ with one of the elements of the set expression $\boldsymbol{s}$:

$$\llparenthesis\ \boldsymbol{n}_1, \ldots \boldsymbol{n}_m : \boldsymbol{s}\ \rrparenthesis^{\mathcal{M}} \quad = \quad \llbracket\ \boldsymbol{s}\ \rrbracket^{\mathcal{M}} \fatsemi \langle\langle \boldsymbol{n}_1{}^{\circ}, \ni\rangle, \ldots, \langle \boldsymbol{n}_m{}^{\circ}, \ni\rangle\rangle \fatsemi \{..\}$$

> **Note:** The simple schema $\boldsymbol{n}_1, \ldots \boldsymbol{n}_m : \boldsymbol{s}$ is well-defined if and only if the expression $\boldsymbol{s}$ 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 $powerT(givenT\,\mathbb{N})$. In this way a schema such as $x : G$ defines the type of x to be $givenT(G)$, as required.

9.5 Schema construction

A schema construction $\boldsymbol{S} \mid \boldsymbol{P}$ is a schema whose signature is that of the schema $\boldsymbol{S}$ and whose components satisfy the constraint of the schema $\boldsymbol{S}$ and the predicate $\boldsymbol{P}$.

Abstract syntax A schema construction is composed from a schema and a predicate.

$$\mathsf{SCONSTRUCTION} ::= \mathsf{SCHEMA} \mid \mathsf{PRED}$$

Concrete form

```
SQBRA Text SQKET
```

Sample representation and transformation

Representation	Abstract
$[D \mid P]$	$[\![D]\!]^{\mathcal{D}} \mid [\![P]\!]^{\mathcal{P}}$
$[D]$	$[\![D]\!]^{\mathcal{D}} \mid \mathbf{true}$

Type The signature of $\langle \boldsymbol{D} \mid \boldsymbol{P} \rangle$ is the same as that of the schema $\boldsymbol{D}$.

$$(\![\ \boldsymbol{S} \mid \boldsymbol{P}\ ]\!)^{\mathcal{T}} \ = \ (\![\ \boldsymbol{S}\ ]\!)^{\mathcal{T}} \cap ((\!\langle\ \boldsymbol{D} \mid \boldsymbol{P}\ \rangle\!)^{\mathcal{T}}) \mathbin{;} (\supseteq)$$

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

$$(\![\ \boldsymbol{D} \mid \boldsymbol{P}\ ]\!)^{\mathcal{M}} \ = \ (\![\ \boldsymbol{D}\ ]\!)^{\mathcal{M}} \cap ((\!\langle\ \boldsymbol{D} \mid \boldsymbol{P}\ \rangle\!)^{\mathcal{M}} \mathbin{;} \ (\supseteq))$$

This is defined only in those environments in which the schema $\boldsymbol{D}$ is defined and when enriched by it result in the predicate $\boldsymbol{P}$ being well-typed.

9.6 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

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

Concrete form

```
NOT Expression
```

Sample representation and transformation

Representation	Abstract
$\neg S$	$\neg[\![S]\!]^{\mathcal{S}}$

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

$$\llparenthesis \neg\ S \rrparenthesis^{\mathcal{T}} \ = \ \llparenthesis S \rrparenthesis^{\mathcal{T}}$$

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:

$$\llparenthesis \neg\ S \rrparenthesis^{\mathcal{M}} \ = \ \llparenthesis S \rrparenthesis^{\mathcal{M}\mathcal{T}} \setminus \llparenthesis S \rrparenthesis^{\mathcal{M}}$$

> **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.

9.7 Schema disjunction

The schema disjunction $S_1 \lor S_2$ 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.

$\mathsf{SDISJUNCTION} ::= \mathsf{SCHEMA} \lor \mathsf{SCHEMA}$

Concrete form

`Expression OR Expression`

Sample representation and transformation

Representation	Abstract
$S_1 \lor S_2$	$[\![S_1]\!]^{\mathcal{S}} \lor [\![S_2]\!]^{\mathcal{S}}$

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

$$\llparenthesis\, S_1 \lor S_2 \,\rrparenthesis^{\mathcal{T}} = \langle \llparenthesis\, S_1 \,\rrparenthesis^{\mathcal{T}}, \llparenthesis\, S_2 \,\rrparenthesis^{\mathcal{T}} \rangle \fatsemi \sqcup$$

Note: The schema disjunction $S_1 \lor S_2$ 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:

$$\llparenthesis\, S_1 \lor S_2 \,\rrparenthesis^{\mathcal{M}} = (\langle \llparenthesis\, S_1 \,\rrparenthesis^{\mathcal{M}\mathcal{T}}, \llparenthesis\, S_2 \,\rrparenthesis^{\mathcal{M}} \rangle \cup \langle \llparenthesis\, S_1 \,\rrparenthesis^{\mathcal{M}}, \llparenthesis\, S_2 \,\rrparenthesis^{\mathcal{M}\mathcal{T}} \rangle) \fatsemi \sqcup$$

9.8 Schema conjunction

Abstract syntax A schema conjunction is composed of two schemas

$$\mathsf{SCONJUNCTION} ::= \mathsf{SCHEMA} \wedge \mathsf{SCHEMA}$$

Concrete form

```
DeclElem SEMICOLON DeclElem {SEMICOLON DeclElem}
Expression AND Expression
```

Sample representation and transformation

Representation	Abstract
$D_1; \ldots; D_n$	$[\![D_1]\!]^{\mathcal{S}} \wedge [\![D_2]\!]^{\mathcal{S}} \wedge \ldots \wedge [\![D_n]\!]^{\mathcal{S}}$
$S_1 \wedge S_2$	$[\![S_1]\!]^{\mathcal{S}} \wedge [\![S_2]\!]^{\mathcal{S}}$

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

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

$$(\!|\, S_1 \wedge S_2 \,|\!)^{\mathcal{T}} = \langle (\!|\, S_1 \,|\!)^{\mathcal{T}}, (\!|\, S_2 \,|\!)^{\mathcal{T}} \rangle \fatsemi \sqcup$$

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:

$$(\!|\, S_1 \wedge S_2 \,|\!)^{\mathcal{M}} = \langle (\!|\, S_1 \,|\!)^{\mathcal{M}}, (\!|\, S_2 \,|\!)^{\mathcal{M}} \rangle \fatsemi \sqcup$$

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

Note: Duplicated schemas 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.

9.9 Schema implication

Abstract syntax A schema implication is composed of two schemas.

$\mathsf{SIMPLICATION} ::= \mathsf{SCHEMA} \Rightarrow \mathsf{SCHEMA}$

Concrete form

`Expression IMPLIES Expression`

Representation	Abstract
$S_1 \Rightarrow S_2$	$[\![S_1]\!]^{\mathcal{S}} \Rightarrow [\![S_2]\!]^{\mathcal{S}}$

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

$$(\![\ S_1 \Rightarrow S_2\ ]\!)^{\mathcal{T}} \ = \ \langle (\![\ S_1\ ]\!)^{\mathcal{T}}, (\![\ S_2\ ]\!)^{\mathcal{T}} \rangle \mathbin{;} \sqcup$$

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 $\neg S_1 \lor S_2$:

$$(\![\ S_1 \Rightarrow S_2\ ]\!)^{\mathcal{M}} \ = \ (\![\ \neg\ S_1 \lor S_2\ ]\!)^{\mathcal{M}}$$

9.10 Schema equivalence

Abstract syntax A schema equivalence is composed of two schemas.

$$\mathsf{SEQUIVALENCE} ::= \mathsf{SCHEMA} \Leftrightarrow \mathsf{SCHEMA}$$

Concrete form

```
Expression IFF Expression
```

Sample representation and transformation

Representation	Abstract
$S_1 \Leftrightarrow S_2$	$[\![S_1]\!]^{\mathcal{S}} \Leftrightarrow [\![S_2]\!]^{\mathcal{S}}$

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

$$(\![\ S_1 \Leftrightarrow S_2\ ]\!)^{\mathcal{T}} \;=\; \langle (\![\ S_1\ ]\!)^{\mathcal{T}}, (\![\ S_2\ ]\!)^{\mathcal{T}} \rangle \fatsemi \sqcup$$

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:

$$(\![\ S_1 \Leftrightarrow S_2\ ]\!)^{\mathcal{M}} \;=\; (\![\ S_1 \Rightarrow S_2 \land S_2 \Rightarrow S_1\ ]\!)^{\mathcal{M}}$$

9.11 Schema projection

The schema projection operator ($\upharpoonright$) hides all 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.

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

Concrete form

```
Expression PROJECTION Expression
```

Sample representation and transformation

Representation	Abstract
$S \upharpoonright T$	$\llbracket S \rrbracket^{\mathcal{S}} \upharpoonright \llbracket T \rrbracket^{\mathcal{S}}$

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):

$$\llparenthesis S_1 \upharpoonright S_2 \rrparenthesis^{\mathcal{T}} = \langle \llparenthesis S_1 \rrparenthesis^{\mathcal{T}}, \llparenthesis S_2 \rrparenthesis^{\mathcal{T}} \rangle \fatsemi \sqcap$$

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:

$$\llparenthesis S_1 \upharpoonright S_2 \rrparenthesis^{\mathcal{M}} = \langle \llparenthesis S_1 \rrparenthesis^{\mathcal{M}}, \llparenthesis S_2 \rrparenthesis^{\mathcal{MT}} \rangle \fatsemi \sqcap$$

> **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.

9.12 Schema hiding

The schema hiding operator ($\setminus$) 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.

$$\mathsf{SHIDING} ::= \mathsf{SCHEMA} \setminus [\mathsf{NAME},\ \mathsf{NAME}, \ldots]$$

Concrete form

```
Expression HIDING BRA NameList1 KET
```

Sample representation and transformation

Representation	Abstract
$S \setminus (n_1, \ldots, n_m)$	$[\![S]\!]^{\mathcal{S}} \setminus [n_1, \ldots, n_m]$

Type The signature of a schema hiding expression is the signature of $\boldsymbol{S}$ with the names from $(\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m)$ removed. Note that $(\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m)$ may contain names not in the signature of *se*:

$$(\!| \ \boldsymbol{S} \setminus [\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m] \ |\!)^{\mathcal{T}} \quad = \quad (\!| \ \boldsymbol{S} \ |\!)^{\mathcal{T}} \mathbin{⨾} (\{\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m\} \mathbin{\lhd})$$

Meaning The value of the schema $\boldsymbol{S}$ in which the components $(\boldsymbol{n}_1, \ldots, \boldsymbol{n}_m)$ have been hidden is the set of bindings which satisfy $\boldsymbol{S}$, with those components removed:

$$(\!| \ \boldsymbol{S} \setminus [\boldsymbol{n}_1, \ldots \boldsymbol{n}_m] \ |\!)^{\mathcal{M}} \quad = \quad (\!| \ \boldsymbol{S} \ |\!)^{\mathcal{M}} \mathbin{⨾} (\{\boldsymbol{n}_1, \ldots \boldsymbol{n}_m\} \lhd)$$

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

9.13 Schema universal quantification

Abstract syntax A schema quantification is constructed from a text-or-expression and a schema.

$$\mathsf{SUNIVQUANT} ::= \forall \mathsf{SCHEMA} \bullet \mathsf{SCHEMA}$$

Concrete form

```
EXISTS TextOrExpression DOT Expression
```

Sample representation and transformation

Representation	Abstract
$\forall\, TE \bullet S$	$\forall [\![\, TE \,]\!]^{\mathcal{TE}} \bullet [\![\, S \,]\!]^{\mathcal{S}}$

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

$$(\!|\, \forall \boldsymbol{TE} \bullet \boldsymbol{S} \,|\!)^{\mathcal{T}} = \langle (\!|\, \boldsymbol{S} \,|\!)^{\mathcal{T}}, (\!|\, \boldsymbol{TE} \,|\!)^{\mathcal{T}} \rangle \mathbin{;} \leftharpoonup$$

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

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

$$(\!|\, \forall \boldsymbol{TE} \bullet \boldsymbol{S} \,|\!)^{\mathcal{M}} = (\!|\, \neg \exists \boldsymbol{TE} \bullet \neg \boldsymbol{S} \,|\!)^{\mathcal{M}}$$

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

9.14 Schema existential quantification

Abstract syntax A schema quantification is composed of a text-or-expression and a schema.

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

Concrete form

```
EXISTS TextOrExpression DOT Expression
```

Sample representation and transformation

Representation	Abstract
$\exists\, TE \bullet S$	$\exists \llbracket TE \rrbracket^{\mathcal{TE}} \bullet \llbracket S \rrbracket^{\mathcal{S}}$

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

$$\llparenthesis\, \exists \boldsymbol{TE} \bullet \boldsymbol{S}\, \rrparenthesis^{\mathcal{T}} \;=\; \langle \llparenthesis\, \boldsymbol{S}\, \rrparenthesis^{\mathcal{T}}, \llparenthesis\, \boldsymbol{TE}\, \rrparenthesis^{\mathcal{T}} \rangle \fatsemi \leftharpoonup$$

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

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

$$\llparenthesis\, \exists \boldsymbol{TE} \bullet \boldsymbol{S}\, \rrparenthesis^{\mathcal{M}} \;=\; \langle \llparenthesis\, \boldsymbol{S}\, \rrparenthesis^{\mathcal{M}}, \llparenthesis\, \boldsymbol{TE}\, \rrparenthesis^{\mathcal{M}} \fatsemi \leftharpoonup$$

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

9.15 Schema unique existential quantification

Abstract syntax A schema quantification is composed of a text-or-expression and a schema.

$$\mathsf{SUNIQUEQUANT} ::= \exists_1 \mathsf{SCHEMA} \bullet \mathsf{SCHEMA}$$

Concrete form

```
EXISTS1 TextOrExpression DOT Expression
```

Sample representation and transformation

Representation	Abstract
$\exists_1 TE \bullet S$	$\exists_1 [\![TE]\!]^{\mathcal{TE}} \bullet [\![S]\!]^{\mathcal{S}}$

Type

$$(\!\lbrack\ \exists_1 \boldsymbol{TE} \bullet \boldsymbol{S}\ \rbrack\!)^{\mathcal{T}} \quad = \quad \langle (\!\lbrack\ \boldsymbol{S}\ \rbrack\!)^{\mathcal{T}}, (\!\lbrack\ \boldsymbol{TE}\ \rbrack\!)^{\mathcal{T}} \rangle \fatsemi \leftharpoonup$$

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

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

$$(\!\lbrack\ \exists_1 \boldsymbol{TE} \bullet \boldsymbol{S}\ \rbrack\!)^{\mathcal{M}} \quad = \quad \text{to be defined}$$

9.16 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.

$$\mathsf{SRENAMING} ::= \mathsf{SCHEMA}\ [\mathsf{NAME/NAME},\ /\mathsf{NAME/NAME}, \ldots]$$

Concrete form

```
Expression SQBRA RenameList SQKET
```

Sample representation and transformation

Representation	Abstract
$S\,[x_1/y_1, x_2/y_2, \ldots x_n/y_n]$	$[\![S]\!]^{\mathcal{S}}[\boldsymbol{x}_1/\boldsymbol{y}_1,\ \boldsymbol{x}_2/\boldsymbol{y}_2,\ \ldots \boldsymbol{x}_n/\boldsymbol{y}_n]$

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

$$(\!|\ \boldsymbol{S}[\boldsymbol{Nl}]\ |\!)^{\mathcal{T}} \;=\; (\!|\ \boldsymbol{S}\ |\!)^{\mathcal{T}} \fatsemi \exists(\langle\!|\ \boldsymbol{Nl}\ |\!\rangle^{\mathcal{N}} \;\otimes\; I)$$

Meaning

$$(\!|\ \boldsymbol{S}[\boldsymbol{Nl}]\ |\!)^{\mathcal{M}} \;=\; (\!|\ \boldsymbol{S}\ |\!)^{\mathcal{M}} \fatsemi \exists(\langle\!|\ \boldsymbol{Nl}\ |\!\rangle^{\mathcal{N}} \;\otimes\; I)$$

> **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.

9.17 Substituted schema

The meaning of the substituted schema $\boldsymbol{b} \odot \boldsymbol{S}$ is the same as the meaning of the schema $\boldsymbol{S}$ in the environment enriched by the binding $\boldsymbol{b}$.

Abstract syntax A substituted schema is composed of an expression and a schema.

$$\mathsf{SSUBSTITUTION} ::= \mathsf{EXP} \odot \mathsf{SCHEMA}$$

Concrete form

```
Expression SUBST Expression
```

Sample representation and transformation

Representation	Abstract
$b \odot S$	$\llbracket(b)\rrbracket^{\mathcal{E}} \odot \llbracket(S)\rrbracket^{\mathcal{S}}$

Type The signature of the substituted schema $\boldsymbol{b} \odot \boldsymbol{S}$ is the signature of the schema $\boldsymbol{S}$ in the type-environment enriched by the binding $\boldsymbol{b}$.

$$\llparenthesis\ \boldsymbol{b} \odot \boldsymbol{S}\ \rrparenthesis^{\mathcal{T}} \ = \ \langle I, \llbracket\ \boldsymbol{b}\ \rrbracket^{\mathcal{T}} \fatsemi schemaT^{-1}\rangle \fatsemi \oplus \fatsemi \llparenthesis\ \boldsymbol{S}\ \rrparenthesis^{\mathcal{T}}$$

A substituted schema is well-typed if and only if the binding is well-typed and the schema is well-typed in the enriched environment.

Meaning The situations of the substituted schema $\boldsymbol{b} \odot \boldsymbol{S}$ are the situations of the schema $\boldsymbol{S}$ in the environment enriched by the binding $\boldsymbol{b}$.

$$\llparenthesis\ \boldsymbol{b} \odot \boldsymbol{S}\ \rrparenthesis^{\mathcal{M}} \ = \ \langle I, \llbracket\ \boldsymbol{b}\ \rrbracket^{\mathcal{M}} \fatsemi \langle \_, \_\rangle\rangle \fatsemi \oplus \fatsemi \llparenthesis\ \boldsymbol{S}\ \rrparenthesis^{\mathcal{M}}$$

Editor's note: Revised versions of the following subsections have been included in Annex F, *The logical theory of Z*:

Free variables
Substitution

□

10 Section

Notes on this section of the Z Standard

Section title: Section
Section editor: Rob Arthan
Source file: section.tex
Most recent update: 12th September 1995
Formatted: 14th September 1995

Note: Several of the early descriptions of Z included modularity features, in which various names were used for the unit of modularity ('chapter', 'document', etc.). The following description is a place-holder for further work on this feature, which will be based on the proposal in the paper "Modules for Z". See reference [2].

10.1 Introduction

The paragraphs making up a specification may be organized as a collection of named sections. These provide a unit of modularity for Z specifications.

Abstract syntax

$$\mathsf{SECTION} ::= \textbf{section}\ \mathsf{NAME}\ \textbf{parents}\ \mathsf{NAME},\ \mathsf{NAME},\ \ldots\ \bullet\ \mathsf{PARAGRAPHS}$$

$$\mathsf{PARAGRAPHS} ::= \mathsf{PAR},\ \mathsf{PAR},\ \ldots$$

Concrete form

```
Section Name [PARENTS Name {COMMA Name} DOT ] Paragraph {SEP Paragraph}
```

□

11 Paragraph

Notes on this section of the Z Standard

Section title: Paragraph
Section editor: Peter Lupton
Source file: par.tex
Most recent update: 13th September 1995
Formatted: 14th September 1995

11.1 Introduction

Each paragraph of Z can: (1) augment the environment by a declaration, and (2) 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; furthermore, any predicates contained within it are true. These environments are related to those which include the new variables declared in the signature of the paragraph and which satisfy any property denoted by the paragraph.

$$\llparenthesis\, \mathsf{PAR} \,\rrparenthesis^{\mathcal{T}} \;:\; Tenv ⇸ Tenv$$

$$\llparenthesis\, \mathsf{PAR} \,\rrparenthesis^{\mathcal{M}} \;:\; Env \longleftrightarrow Env$$

We can prove the following

$$\vdash \llparenthesis\, \boldsymbol{Par} \,\rrparenthesis^{\mathcal{M}} ⨟ \Upsilon \;\subseteq\; \Upsilon ⨟ \llparenthesis\, \boldsymbol{Par} \,\rrparenthesis^{\mathcal{T}}$$

Abstract Syntax

$$\begin{array}{lcl} \mathsf{PAR} & ::= & \mathsf{GIVENSETDEF} \\ & | & \mathsf{GLOBALPRED} \\ & | & \mathsf{GLOBALSCHEMA} \\ & | & \mathsf{GENERICSCHEMA} \\ & | & \mathsf{GLOBALDEF} \\ & | & \mathsf{GENERICDEF} \\ & | & \mathsf{FREETYPEDEF} \\ & | & \mathsf{CONJECTURE} \end{array}$$

11.2 Given set

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

> **Note:** Distinctly named given sets have distinct types.

Abstract syntax

GIVENSETDEF ::= **given** [NAME, NAME, ...]

Concrete form

`SQBRA NameList1 SQKET`

Sample representation and transformation

Representation	Abstract
$[X_1, \ldots, X_n]$	**given** $[\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n]$

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 over-ride the environment.

Note that a given set definition of $\mathbb{N}$ results in $\mathbb{N}$ having the type: *powerT givenT* $\mathbb{N}$.

$$\llbracket\, \mathbf{given}\, [\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n] \,\rrbracket^{\mathcal{T}} = \langle I, (\{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\} \lhd \mathit{givenT} \mathbin{;} \mathit{powerT})^{\circ} \rangle \mathbin{;} (\oplus)$$

Meaning To enrich the meaning environment, we construct a binding of the given set names (those in $\operatorname{ran} s$) to typed values in the world of sets. For this approach 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.

$$\llbracket\, \mathbf{given}\, [\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n] \,\rrbracket^{\mathcal{M}} = \langle I, (\{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\} \lhd \mathit{givenT} \mathbin{;} \langle \mathit{powerT}, \mathit{Carrier} \rangle)^{\circ} \rangle \mathbin{;} (\oplus)$$

11.3 Global declaration

A global declaration takes the form of a schema, whose signature is to be used to augment the environment. A global declaration introduces variables and specifies further properties of the elements denoted by them.

Abstract syntax

> **Note:** The following definition of Global declaration deals with cases where the DeclPart is non-null. For cases where DeclPart is null, see 11.4 below.

$$\mathsf{GLOBALSCHEMA} ::= \mathbf{def}\ \mathsf{SCHEMA}$$

Concrete form

`AX BasicDecl [ST Predicate] END`

`AX Expression [ST Predicate] END`

Sample representation and transformation

Representation	Abstract
AX D ST P END	$\mathbf{def}\ [\![D]\!]^{\mathcal{S}} \mid [\![P]\!]^{\mathcal{P}}$
AX D END	$\mathbf{def}\ [\![D]\!]^{\mathcal{S}}$
AX S ST P END	$\mathbf{def}\ [\![S]\!]^{\mathcal{E}} \mid [\![P]\!]^{\mathcal{P}}$
AX S END	$\mathbf{def}\ [\![S]\!]^{\mathcal{E}}$

The abstract form of a global declaration is a pair of paragraphs, one containing a declaration and the other a predicate. If the AxiomPart is omitted the abstract form is one declaration paragraph.

Type When new variables are declared, the environment is enriched by a function from their names to their meanings. 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:

$$\langle\!|\ \mathbf{def}\ \boldsymbol{S}\ |\!\rangle^{\mathcal{T}} = \langle\!|\ \boldsymbol{S}\ |\!\rangle^{\mathcal{T}}$$

Meaning

$$\langle\!|\ \mathbf{def}\ \boldsymbol{S}\ |\!\rangle^{\mathcal{M}} = \langle\!|\ \boldsymbol{S}\ |\!\rangle^{\mathcal{M}}$$

11.4 Constraint

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

$$\mathsf{GLOBALPRED} ::= \textbf{where}\ \mathsf{PRED}$$

Concrete form

```
AX [ST Predicate] END
```

Sample representation and transformation

Representation	Abstract
P	**where** $[\![P]\!]^{\mathcal{P}}$
`AX ST` P `END`	**where** $[\![P]\!]^{\mathcal{P}}$
`AX END`	**where true**

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:

$$\langle\!|\ \boldsymbol{P}\ |\!\rangle^{\mathcal{T}} \;=\; \boldsymbol{I} \rhd \{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{T}}$$

Meaning For meaning environment:

$$\langle\!|\ \boldsymbol{P}\ |\!\rangle^{\mathcal{M}} \;=\; \boldsymbol{I} \rhd \{\![\ \boldsymbol{P}\ ]\!\}^{\mathcal{M}}$$

11.5 Generic declaration

A generic declaration 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

$$\mathsf{GENERICSCHEMA} \; ::= \; \mathbf{gendef}\,[\mathsf{NAME},\ \mathsf{NAME},\ \ldots]\;\mathbf{const}\ \mathsf{SCHEMA}$$

Concrete form

```
NAME Formals DEFINE_EQUAL Expression
SCH Name [Formals] IS [DeclPart] [ST Predicate] END
```

Sample representation and transformation

Representation	Abstract
GEN $[X_1, \ldots, X_n]$ BAR D ST P END	$\mathbf{gendef}\,\langle \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\rangle\ \mathbf{const}\ [\![D]\!]^{\mathcal{S}} \quad \mid\ [\![P]\!]^{\mathcal{P}}$
GEN $[X_1, \ldots, X_n]$ BAR D END	$\mathbf{gendef}\,\langle \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\rangle\ \mathbf{const}\ [\![D]\!]^{\mathcal{S}}$
GEN $[X_1, \ldots, X_n]$ BAR S ST P END	$\mathbf{gendef}\,\langle \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\rangle\ \mathbf{const}\ [\![S]\!]^{\mathcal{S}} \quad \mid\ [\![P]\!]^{\mathcal{P}}$
GEN $[X_1, \ldots, X_n]$ BAR S END	$\mathbf{gendef}\,\langle \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\rangle\ \mathbf{const}\ [\![S]\!]^{\mathcal{S}}$
GEN $[X_1, \ldots, X_n]$ BAR END	$\mathbf{gendef}\,\langle \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\rangle\ \mathbf{const}\ \mathbf{emptyschema}$

Editor's note: The semantic definitions in the following pages are adapted from the paper 'Generic Lifting' by Peter Lupton, available on the ftp server in the subdirectory 'papers'. Subsidiary definitions from this paper, used in the following definition of generic lifting, will be added to the early chapters of the standard.

Type The type of a generic declaration is given by the generic lifting of the type of the schema defined by the generic declaration.

$$\langle\!\!|\ \mathbf{gendef}\,\langle \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\rangle\ \mathbf{const}\ \boldsymbol{S}\ |\!\!\rangle^{\mathcal{T}} = \langle I\,,\ {}^{(X_1,\ldots,X_n)}\,(\langle\!\!|\ \boldsymbol{S}\ |\!\!\rangle^{\mathcal{T}})\rangle\ ;\ (\oplus)$$

Meaning The meaning of a generic declaration is given by the generic lifting of the meaning of the schema defined by the generic declaration.

$$\llparenthesis\ \mathbf{gendef}\,\langle \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n \rangle\,\mathbf{const}\ \boldsymbol{S}\ \rrparenthesis^{\mathcal{M}} = \langle I\,,\ {}^{(X_1,\ldots,X_n)}(\llbracket\ \boldsymbol{S}\ \rrbracket^{\mathcal{M}})\rangle\ ;\ (\oplus)$$

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 declarations are not allowed. The generic declaration 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 declaration are local to the declaration, but they can be instantiated by elements of set type when the generic variable is used.

A generic declaration does not give a single type: rather, a function from the generic parameters to types is defined.

Let X and Y be generic formal parameters and consider a generic declaration which 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 declarations do not create relationships between the type of their formal parameters. Such relationships can always be eliminated within a specification.

Since all the type constructors are bijections, any 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 instance, for $x \in y$ the 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 $y : X$.

11.6 Global definition

A global definition introduces a single name. The value given to that name might be non-unique, in which case the global definition will relate a single environment meaning environment to many.

Abstract syntax

$$\mathsf{GLOBALDEF} ::= \mathbf{abbr}\,\mathsf{NAME} := \mathsf{EXP}$$

Concrete form

```
SCH Name IS [DeclPart] [ST Predicate] END
NAME DEFINE_EQUAL Expression
```

Sample representation and transformation

Representation	Abstract
N `DEFINE_EQUAL` E	$\mathbf{abbr}\,\mathrm{N} := [\![E]\!]^{\mathcal{E}}$
`SCH` N `IS` D `ST` P `END`	$\mathbf{abbr}\,\mathrm{N} := [\![D]\!]^{\mathcal{S}} \mid [\![P]\!]^{\mathcal{P}}$
`SCH` N `IS` D `END`	$\mathbf{abbr}\,\mathrm{N} := [\![D]\!]^{\mathcal{S}} \mid \mathbf{true}$
`SCH` N `IS` S `ST` P `END`	$\mathbf{abbr}\,\mathrm{N} := [\![S]\!]^{\mathcal{E}} \mid [\![P]\!]^{\mathcal{P}}$
`SCH` N `IS` S `END`	$\mathbf{abbr}\,\mathrm{N} := [\![S]\!]^{\mathcal{E}}$
`SCH` N `IS ST` P `END`	$\mathbf{abbr}\,\mathrm{N} := \mathbf{emptyschema} \mid [\![P]\!]^{\mathcal{P}}$
`SCH` N `IS END`	$\mathbf{abbr}\,\mathrm{N} := \mathbf{emptyschema}$

Type A global definition adds the name defined to the type environment and this name is mapped to its type.

$$\{\!|\ \mathbf{abbr}\boldsymbol{N} \mathrel{\widehat{=}} \boldsymbol{X}\ |\!\}^{\mathcal{T}} \;=\; \langle I,\ \langle \boldsymbol{N}^{\circ}, [\![\ \boldsymbol{X}\ ]\!]^{\mathcal{T}}\rangle \mathbin{⨾} \{-\}\rangle \mathbin{⨾} (\oplus)$$

Meaning A global definition adds the name defined to the environment and this name is mapped to each possible meaning.

$$\{\!|\ \mathbf{abbr}\boldsymbol{N} \mathrel{\widehat{=}} \boldsymbol{X}\ |\!\}^{\mathcal{M}} \;=\; \langle I,\ \langle \boldsymbol{N}^{\circ}, [\![\ \boldsymbol{X}\ ]\!]^{\mathcal{M}}\rangle \mathbin{⨾} \{-\}\rangle \mathbin{⨾} (\oplus)$$

11.7 Generic definition

A generic definition adds a single name to the environment as a generically defined name. That is, all possible values of the name's generic parameters are used to evaluate the expression being defined.

Abstract syntax

$$\mathsf{GENERICDEF} ::= \mathbf{abbr}\ [\mathsf{NAME},\ \mathsf{NAME},\ \ldots] := \mathsf{EXP}$$

Concrete form

```
NAME Formals DEFINE_EQUAL Expression
SCH Name [Formals] IS [DeclPart] [ST Predicate] END
```

Sample representation and transformation

Representation	Abstract
$N\ [X_1, \ldots, X_n]$ `DEFINE_EQUAL` X	$\mathbf{abbr}\,\mathrm{N}\,[X_1, \ldots, X_n] := [\![X]\!]^{\mathcal{E}}$
`SCH` $N\ [X_1, \ldots, X_n]$ `IS` D `ST` P `END`	$\mathbf{abbr}\,\mathrm{N}\,[X_1, \ldots, X_n] := [\![D]\!]^{\mathcal{S}} \mid [\![P]\!]^{\mathcal{P}}$
`SCH` $N\ [X_1, \ldots, X_n]$ `IS` D `END`	$\mathbf{abbr}\,\mathrm{N}\,[X_1, \ldots, X_n] := [\![D]\!]^{\mathcal{S}}$
`SCH` $N\ [X_1, \ldots, X_n]$ `IS` S `ST` P `END`	$\mathbf{abbr}\,\mathrm{N}\,[X_1, \ldots, X_n] := [\![S]\!]^{\mathcal{E}} \mid [\![P]\!]^{\mathcal{P}}$
`SCH` $N\ [X_1, \ldots, X_n]$ `IS` S `END`	$\mathbf{abbr}\,\mathrm{N}\,[X_1, \ldots, X_n] := [\![S]\!]^{\mathcal{E}}$
`SCH` $N\ [X_1, \ldots, X_n]$ `END`	$\mathbf{abbr}\,\mathrm{N}\,[X_1, \ldots, X_n] := \mathbf{emptyschema}$

Type The type of a generic definition is given by the generically lifted type of the schema declaration formed by the name and the expression defined by the generic definition:

$$\langle\!|\ \mathbf{abbr}\,\boldsymbol{N}\,[\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n] := \boldsymbol{E}\ |\!\rangle^{\mathcal{T}} = \langle I,\ ^{(X_1, \ldots, X_n)}((\langle\!|\ \boldsymbol{N} : \boldsymbol{E}\ |\!\rangle^{\mathcal{T}})^{\mathcal{T}} \mathbin{⨟} (\oplus)$$

Meaning The meaning of a generic definition is