INTERNATIONAL STANDARD

ISO/IEC 13568

First edition
2002-07-01

Information technology — Z formal specification notation — Syntax, type system and semantics

Technologies de l'information — Notation Z pour la spécification formelle — Syntaxe, système de caractères et sémantique

Reference number
ISO/IEC 13568:2002(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

Contents

Figures

Tables

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 13568 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Annexes A to C form a normative part of this International Standard. Annexes D and E are for information only.

Introduction

This International Standard specifies the syntax, type system and semantics of the Z notation, as used in formal specification.

A specification of a system should aid understanding of that system, assisting development and maintenance of the system. Specifications need express only abstract properties, unlike implementations such as detailed algorithms, physical circuits, etc. Specifications may be loose, allowing refinement to many different implementations. Such abstract and loose specifications can be written in Z notation.

A specification written in Z notation models the specified system: it names the components of the system and expresses the constraints between those components. The meaning of a Z specification—its semantics—is defined as the set of interpretations (values for the named components) that are consistent with the constraints.

Z uses mathematical notation, hence specifications written in Z are said to be formal: the meaning is captured by the form of the mathematics used, independent of the names chosen. This formal basis enables mathematical reasoning, and hence proofs that desired properties are consequences of the specification. The soundness of inference rules used in such reasoning should be proven relative to the semantics of the Z notation.

This International Standard establishes precise syntax and semantics for a system of notation for mathematics, providing a basis on which further mathematics can be formalized.

Particular characteristics of Z include:

- its extensible toolkit of mathematical notation;
- its schema notation for specifying structures in the system and for structuring the specification itself; and
- its decidable type system, which allows some well-formedness checks on a specification to be performed automatically.

Examples of the kinds of systems that have been specified in Z include:

- safety critical systems, such as railway signalling, medical devices, and nuclear power systems;
- security systems, such as transaction processing systems, and communications; and
- general systems, such as programming languages and floating point processors.

Standard Z will also be appropriate for use in:

- formalizing the semantics of other notations, especially in standards documents.

This is the first ISO standard for the Z notation. Much has already been published about Z. Most uses of the Z notation have been based on the examples in the book "Specification Case Studies" edited by Hayes [2][3]. Early definitions of the notation were made by Sufrin [14] and by King *et al* [8]. Spivey's doctoral thesis showed that the semantics of the notation could be defined in terms of sets of models in ZF set theory [11]. His book "The Z Notation—A Reference Manual" [12][13] is the most complete definition of the notation, prior to this International Standard. Differences between Z as defined here and as defined in [13] are discussed in [15]. This International Standard addresses issues that have been resolved in different ways by different users, and hence encourages interchange of specifications between diverse tools. It also aims to be a complete formal definition of Z.

Information technology— Z formal specification notation— Syntax, type system and semantics

1 Scope

The following are within the scope of this International Standard:

- the syntax of the Z notation;
- the type system of the Z notation;
- the semantics of the Z notation;
- a toolkit of widely used mathematical operators;
- LaTeX [10] and e-mail mark-ups of the Z notation.

The following are outside the scope of this International Standard:

- any method of using Z, though an informative annex (E) describes one widely-used convention.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 10646-1:2000, *Information technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane, with amendment 1, with its amendments and corrigenda*

ISO/IEC 10646-2:2001, *Information technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 2: Supplementary Planes*

ISO/IEC 14977:1996, *Information technology—Syntactic metalanguage—Extended BNF*

3 Terms and definitions

For the purposes of this International Standard, the following terms and definitions apply. *Italicized* terms in definitions are themselves defined in this list.

3.1
binding
finite function from names to values

3.2
capture
cause a reference expression to refer to a different declaration from that intended

3.3
carrier set
set of all values in a type

3.4
constraint
property that is either true or false

3.5
environment
function from names to information used in type inference

3.6
interpretation
function from global names of a section to values in the *semantic universe*

3.7
metalanguage
language used for defining another language

3.8
metavariable
name denoting an arbitrary phrase of a particular syntactic class

3.9
model
interpretation that makes the defining *constraints* of the corresponding section be true

3.10
schema
set of *bindings*

3.11
scope of a declaration
part of a specification in which a reference expression whose name is the same as a particular declaration refers to that declaration

3.12
scope rules
rules determining the *scope of a declaration*

3.13
semantic universe
set of all semantic values, providing representations for both non-generic and generic Z values

3.14
signature
function from names to types

3.15
type universe
set of all type values, providing representations for all Z types

3.16
Z core language
Z notation defined by this International Standard excepting the notation of the mathematical toolkit

3.17
ZF set theory
Zermelo-Fraenkel set theory

4 Metalanguages

The metalanguages are the notations used to define the syntax, type system and semantics of Z.

4.1 Syntactic metalanguage

The syntactic metalanguage used is the subset of the standard ISO/IEC 14977:1996 [7] summarised in Table 1, with modifications so that the mathematical symbols of Z can be presented in a more comprehensible way.

Table 1 — Syntactic metalanguage

Symbol	Definition
=	defines a non-terminal on the left in terms of the syntax on the right.
|	separates alternatives.
,	separates notations to be concatenated.
—	separates notation on the left from notation to be excepted on the right.
{ }	delimit notation to be repeated zero or more times.
[ ]	delimit optional notation.
( )	are grouping brackets (parentheses).
' '	delimit a terminal symbol.
;	terminates a definition.
? ?	delimit informal definition of notation.
(* *)	delimit commentary.

The infix operators | and , have precedence such that parentheses are needed when concatenating alternations, but not when alternating between concatenations. The exception notation is always used with parentheses, making its precedence irrelevant. Whitespace separates tokens of the syntactic metalanguage; it is otherwise ignored.

EXAMPLE The lexis of a `NUMERAL` token, and its informal reading, are as follows.

```
NUMERAL             = NUMERAL , DECIMAL
                    | DECIMAL
                    ;
```

The non-terminal symbol `NUMERAL` stands for a maximal sequence of one or more decimal digit characters (without intervening white space).

The changes to ISO/IEC 14977:1996 allow use of mathematical symbols in the names of non-terminals, and are formally defined as follows.

```
Meta identifier character = ? all cases from ISO/IEC 14977:1996 ?
                          | '(' | ')' | '[' | ']' | '{' | '}' | '⦇' | '⦈' | '⟪' | '⟫'
                          | 'ℙ' | ':' | ',' | '|' | '&' | '\' | '/' | '.' | ';' | '_' | '='
                          | '?' | '⊢' | '∀' | '•' | '∃' | '⇔' | '⇒' | '∨' | '∧' | '¬'
                          | '∈' | '↾' | '×' | 'λ' | 'μ' | 'θ' | '₉' | '≫' | '₈'
                          ;
```

NOTE This anticipates a future version of ISO/IEC 14977:1996 permitting use of the characters of ISO/IEC 10646 [5][6].

The following standard metalanguage characters have been overloaded as `Meta identifier characters`: `'='`, `'|'`, `','`, `'{'`, `'}'`, `'['`, `']'`, `'('`, `')'` and `';'`. Uses of them as `Meta identifiers` are with the common

suffix -tok, e.g. (-tok, which may be viewed as a postfix metalanguage operator. Uses of them as meta-operators are further distinguished by preceding and following these uses with space.

A further change to ISO/IEC 14977:1996 is the use of multiple fonts: metalanguage characters and non-terminals are in `typewriter`, those non-terminals that correspond to Z tokens appear as those Z tokens normally appear, typically in Roman, and comments are in *italic*.

The syntactic metalanguage is used in defining Z characters, lexis, concrete syntax and annotated syntax (clauses 6, 7, 8 and 10).

4.2 Mathematical metalanguage

4.2.1 Introduction

Logic and ZF set theory are the basis for the semantics of Z. In this section the specific notations used are described. The notations used here are deliberately similar in appearance to those of Z itself, but are grounded only on the logic and set theory developed by the wider mathematical community.

The mathematical metalanguage is used in type inference rules and in semantic relations (clauses 13 and 15).

4.2.2 Parentheses

The forms of proposition and expression are given below. Where there could be any ambiguity in the parsing, usually parentheses have been used to clarify, but in any other case the precedence conventions of Z itself are intended to be used.

The use of parentheses is given in tabular form in Table 2, where p stands for any proposition and e stands for any expression.

Table 2 — Parentheses in metalanguage

Notation	Definition
(p)	p
(e)	e

The same brackets symbols are used around pairs and tuple extensions (Table 12), but those cannot be omitted in this way.

4.2.3 Propositions

4.2.3.1 Introduction

The value of a metalanguage proposition is either *true* or *false*. The values *true* and *false* are distinct. In this International Standard, no proposition of the metalanguage is both *true* and *false*; that is, this metatheory is consistent. Furthermore, every proposition is either *true* or *false*, even where it is not possible to say which; that is, the logic is two-valued.

4.2.3.2 Propositional connectives

The propositional connectives of negation, conjunction and disjunction are used. In Table 3 and later, p, p_2, etc, represent arbitrary propositions.

Conjunction is also sometimes indicated by writing propositions on successive lines, as a vertical list.

4.2.3.3 Quantifiers

Existential, universal and unique-existential quantifiers are used. In Tables 4 and 5 and later, i, i_2, etc, represent arbitrary names, e, e_2, etc, represent arbitrary expressions, and ... represents zero or more repetitions of the surrounding formulae; in these tables, the propositions can contain references to the names, but the expressions cannot.

Table 3 — Propositional connectives in metalanguage

Notation	Name	Definition
$\neg\ p$	negation	*true* iff p is *false*
$p_1 \wedge p_2$	conjunction	*true* iff p_1 and p_2 are both *true*
$p_1 \vee p_2$	disjunction	*false* iff p_1 and p_2 are both *false*

Table 4 — Quantifiers in metalanguage

Notation	Name	Definition
$\exists\ i_1 : e_1;\ ...;\ i_n : e_n \bullet p$	existential quantification	there exist ($\exists$) values of i_1 in set e_1, ..., i_n in set e_n ($i_1 : e_1;\ ...;\ i_n : e_n$) such that ($\bullet$) p is *true*
$\forall\ i_1 : e_1;\ ...;\ i_n : e_n \bullet p$	universal quantification	for all ($\forall$) values of i_1 in set e_1, ..., i_n in set e_n ($i_1 : e_1;\ ...;\ i_n : e_n$), it is true that ($\bullet$) p is *true*
$\exists_1\ i_1 : e_1;\ ...;\ i_n : e_n \bullet p$	unique existential quantification	there exists exactly one ($\exists_1$) configuration of values i_1 in set e_1, ..., i_n in set e_n ($i_1 : e_1;\ ...;\ i_n : e_n$) such that ($\bullet$) p is *true*

Certain abbreviations in the writing of quantifications are permitted, as given in Table 5. Some of their expansions involve lesser abbreviations.

Table 5 — Abbreviations in quantifications in metalanguage

Notation	Definition
$i_1, i_2, ..., i_n : e$	$i_1 : e;\ i_2, ..., i_n : e$
$\exists\ i_1, i_2, ..., i_n : e \mid p_1 \bullet p_2$	$\exists\ i_1, i_2, ..., i_n : e \bullet p_1 \wedge p_2$
$\forall\ i_1, i_2, ..., i_n : e \mid p_1 \bullet p_2$	$\forall\ i_1, i_2, ..., i_n : e \bullet (\neg\ p_1) \vee p_2$
$\exists_1\ i_1, i_2, ..., i_n : e \mid p_1 \bullet p_2$	$\exists_1\ i_1, i_2, ..., i_n : e \bullet p_1 \wedge p_2$

4.2.3.4 Conditional expression

The conditional expression allows the choice between two alternative values according to the truth or falsity of a given proposition, as defined in Table 6.

Table 6 — Conditional expression in metalanguage

Notation	Definition
if p then e_1 else e_2	either p is *true* and e_1 is the value, or p is *false* and e_2 is the value

4.2.4 Sets

4.2.4.1 Introduction

The notation used here is based on ZF set theory, as described in for example [1], and the presentation here is guided by the order given there. In that theory there are only sets. Members of sets can be only other sets. The word "element" may be used loosely when referring to set members treated as atomic, without regard to their set nature. If metalanguage operations are applied to inappropriate arguments, they produce unspecified results rather than being undefined.

4.2.4.2 The universe

The universe, $\mathbb{U}$, denotes a world of sets, providing semantic values for Z expressions. $\mathbb{U}$ is big enough to contain the set NAME from which Z names are drawn, and an infinite set. Also, $\mathbb{U}$ is closed under formation of powersets and products. The formation of a suitable $\mathbb{U}$ comprising models of sets and tuples, as needed to model Z, is well-known in ZF set theory and is assumed in this International Standard. A Z binding is modelled by a ZF set of pairs of NAME and value. A Z generic is modelled by a ZF set of pairs (a function) from sets (the generic arguments) to a value (the instantiated generic).

4.2.4.3 Propositions about sets and elements

The simplest propositions about sets are the relationships of membership, non-membership, subset and equality between sets or their elements, as detailed in Table 7.

Table 7 — Propositions about sets in metalanguage

Notation	Name	Definition
$e_1 \in e_1$	membership	*true* iff e_1 is a member of set e_2
$e_1 \notin e_2$	non-membership	$\neg\ e_1 \in e_2$
$e_1 \subseteq e_2$	subset	$\forall\ i : e_1 \bullet i \in e_2$
$e_1 = e_2$	equality	for e_1 and e_2 considered as sets, $e_1 \subseteq e_2 \land e_2 \subseteq e_1$

4.2.4.4 Basic set operations

ZF set theory constructs its repertoire of set operations starting with the axiom of empty set, then showing how to build up sets using the axioms of pairing and of union, and how to trim them back with the axiom of subset or separation.

For the purposes of this mathematical metalanguage, the simplest form of set comprehension is defined directly using the axiom of separation in Table 8. The existence of a universal set $\mathbb{U}$ is assumed. Other forms of set comprehension are defined in terms of the simplest form, using rules in which i is any name distinct from those already in use. Also defined in Table 8 are notations for empty set, finite set extensions, unions, intersections and differences.

Table 8 — Basic set operations in metalanguage

Notation	Name	Definition
$\{i : e \mid p\}$	set comprehension	subset of elements i of e such that p, by axiom of separation
$\{i_1 : e_1;\ ...;\ i_n : e_n \bullet e\}$	set comprehension	$\{i : \mathbb{U} \mid \exists\ i_1 : e_1;\ ...;\ i_n : e_n \bullet i = e\}$
$\{i_1 : e_1;\ ...;\ i_n : e_n \mid p \bullet e\}$	set comprehension	$\{i : \mathbb{U} \mid \exists\ i_1 : e_1;\ ...;\ i_n : e_n \mid p \bullet i = e\}$
$\varnothing$	empty set	$\{i : \mathbb{U} \mid \mathit{false}\}$
$\{\}$	empty set	$\{i : \mathbb{U} \mid \mathit{false}\}$
$\{e\}$	singleton set	$\{i : \mathbb{U} \mid i = e\}$
$e_1 \cup e_2$	union	$\{i : \mathbb{U} \mid i \in e_1 \lor i \in e_2\}$
$\{e_1, e_2, ..., e_n\}$	set extension	$\{e_1\} \cup \{e_2, ..., e_n\}$
$e_1 \cap e_2$	intersection	$\{i : e_1 \mid i \in e_2\}$
$e_1 \setminus e_2$	difference	$\{i : e_1 \mid i \notin e_2\}$

4.2.4.5 Powersets

The axiom of powers asserts the existence of a powerset, which is the set of all subsets of a set. The set of all finite subsets is a subset of the powerset. It is the smallest set containing the empty set and all singleton subsets of e and closed under the operation of forming the union with singleton subsets of e. Their forms are given in Table 9.

Table 9 — Powerset in metalanguage

Notation	Name	Definition
$\mathbb{P}\ e$	set of all subsets	$\{i : \mathbb{U} \mid i \subseteq e\}$
$\mathbb{F}\ e$	set of all finite subsets	$\{i_1 : \mathbb{U} \mid \forall\, i_2 : \mathbb{P}\,\mathbb{P}\, e \mid \varnothing \in i_2 \land (\forall\, i_3 : i_2 \bullet \forall\, i_4 : e \bullet i_3 \cup \{i_4\} \in i_2) \bullet i_1 \in i_2\}$

4.2.5 Natural numbers

Natural numbers are not primitive in ZF set theory, but there are several well established ways of representing them. The choice of coding is irrelevant here and so is not specified. There are notations to measure the cardinality of a finite set, to define addition of natural numbers and to form the set of natural numbers between two stated natural numbers, as given in Table 10.

Table 10 — Operations on natural numbers in metalanguage

Notation	Definition
$e_1 + e_2$	sum of natural numbers e_1 and e_2
$\#\ e$	cardinality of finite set e
$e_1 \mathinner{..} e_2$	set of natural numbers between e_1 and e_2 inclusive

4.2.6 Names

Names are needed for this International Standard. There are several ways of representing names in ZF set theory. The choice of coding is irrelevant here and so is not specified. Only one operation is needed on names; it is an infix operation with highest precedence, and is defined in Table 11.

Table 11 — Decorations of names in metalanguage

Notation	Definition
$i\ decor^{+}$	the name that is like i but with the extra stroke $^{+}$

4.2.7 Tuples and Cartesian products

Tuples and Cartesian products are not primitive in ZF set theory, but there are various ways in which they may be represented within that theory, such as the well-known encoding given by Kuratowski [1]. The choice of coding is irrelevant here and so is not specified. In this International Standard, the syntactic context of the metalanguage expression denoting a value always determines whether or not that value is to be interpreted as an encoding of a tuple. Therefore there is never any possibility of accidental confusion between the encoding used to represent the tuple and any other value that is not a tuple.

In this mathematical metalanguage, tuples and Cartesian products with more than two components are interpreted as nested binary tuples and products, unlike in Z.

The syntactic forms are given in Table 12.

The brackets delimiting a pair or tuple extension written with commas shall not be omitted—they are not grouping parentheses.

4.2.8 Function comprehensions

Table 13 defines the notation for λ, which is a form of comprehension convenient when defining functions.

4.2.9 Relations

A relation is defined to be a set of pairs. There are several operations involving relations, which are given equivalences in Table 14. A proposition about relations is given in Table 15.

Table 12 — Tuples and Cartesian products in metalanguage

Notation	Name	Definition
(e_1, e_2)	pair	
$e_1 \mapsto e_2$	maplet	(e_1, e_2)
$first\ e$		$first(e_1, e_2) = e_1$
$second\ e$		$second(e_1, e_2) = e_2$
$e_1 \times e_2$		$\{i_1 : e_1;\ i_2 : e_2 \bullet (i_1, i_2)\}$
$(e_1, e_2, ..., e_n)$	tuple extension	$(e_1, (e_2, ..., e_n))$ where $n > 2$
$e_1 \times e_2 \times ... \times e_n$	Cartesian product	$e_1 \times (e_2 \times ... \times e_n)$ where $n > 2$
$e \uparrow 1$		e
$e \uparrow n$	iterated product	$e \times ... \times e$ where there are $n \geq 2$ occurrences of e

Table 13 — Function comprehensions in metalanguage

Notation	Definition
$\lambda\ i : e_1 \bullet e_2$	$\{i : e_1 \bullet i \mapsto e_2\}$
$\lambda\ i : e_1 \mid p \bullet e_2$	$\{i : e_1 \mid p \bullet i \mapsto e_2\}$
$\lambda\ i_1 : e_1;\ ...;\ i_n : e_n \bullet e$	$\{i_1 : e_1;\ ...;\ i_n : e_n \bullet (i_1, ..., i_n) \mapsto e\}$
$\lambda\ i_1 : e_1;\ ...;\ i_n : e_n \mid p \bullet e$	$\{i_1 : e_1;\ ...;\ i_n : e_n \mid p \bullet (i_1, ..., i_n) \mapsto e\}$

Table 14 — Relations in metalanguage

Notation	Name	Definition
$id\ e$	identity function	$\lambda\ i : e \bullet i$
$dom\ e$	domain	$\{i : e \bullet first\ i\}$
$e_1 \lhd e_2$	domain restriction	$\{i : e_2 \mid first\ i \in e_1\}$
$e_1 \lhd\!\!\!\!- e_2$	domain subtraction	$\{i : e_2 \mid first\ i \notin e_1\}$
$e_1(\!\mid e_2 \mid\!)$	relational image	$\{i : e_1 \mid first\ i \in e_2 \bullet second\ i\}$
$e_1 \fatsemi e_2$	relational composition	$\{i_1 : e_1;\ i_2 : e_2 \mid second\ i_1 = first\ i_2 \bullet first\ i_1 \mapsto second\ i_2\}$
$e_1 \oplus e_2$	relational overriding	$((dom\ e_2) \lhd\!\!\!\!- e_1) \cup e_2$

Table 15 — Proposition about relations in metalanguage

Notation	Name	Definition
$e_1 \approx e_2$	compatible relations	$(dom\ e_2) \lhd e_1 = (dom\ e_1) \lhd e_2$

4.2.10 Functions

A function is a particular form of relation, where each domain element has only one corresponding range element. Table 16 shows the various forms of function that are identified, each being a set of functions.

Table 16 — Functions in metalanguage

Notation	Name	Definition
$e_1 \nrightarrow e_2$	functions	$\{i_1 : \mathbb{P}(e_1 \times e_2) \mid \forall\ i_2, i_3 : i_1 \mid first\ i_2 = first\ i_3 \bullet second\ i_2 = second\ i_3\}$
$e_1 \rightarrow e_2$	total functions	$\{i : e_1 \nrightarrow e_2 \mid dom\ i = e_1\}$
$e_1 \twoheadrightarrow e_2$	finite functions	$\{i : \mathbb{F}(e_1 \times e_2) \mid i \in e_1 \nrightarrow e_2\}$

4.2.10.1 Application

A function can be juxtaposed with an argument to produce a result, using the notation of Table 17. Metalanguage notations introduced above that match the $e_1\ e_2$ pattern, such as $dom\ e$, are not applications in this sense.

Table 17 — Application in metalanguage

Notation	Name	Definition
$e_1\ e_2$	application	if there exists a unique e_3 such that $e_2 \mapsto e_3$ is in e_1, then the value of $e_1\ e_2$ is e_3, otherwise each $e_1\ e_2$ has a fixed but unknown value

4.2.11 Sequences

A sequence is a particular form of function, where the domain elements are all the natural numbers from 1 to the length of the sequence.

Table 18 — Sequences in metalanguage

Notation	Name	Definition
$\langle e_1, ..., e_n \rangle$	sequence	$\{1 \mapsto e_1, ..., n \mapsto e_n\}$

4.2.12 Disjointness

A labelled family of sets is disjoint when any distinct pair yields sets with no members in common.

Table 19 — Disjointness in metalanguage

Notation	Name	Definition
$disjoint\ e$	disjointness	$\forall\ e_1, e_2 : dom\ \ e \mid e_1 \neq e_2 \bullet e\ e_1 \cap e\ e_2 = \varnothing$

4.3 Transformation metalanguage

Transformation rules map parse trees of phrases to other parse trees. Each transformation rule is written in the following form.

concrete phrase template $\Longrightarrow$ *less concrete phrase template*

EXAMPLE 1 The syntactic transformation rule for a schema definition paragraph, and an informal reading of it, are as follows.

$$\texttt{SCH}\ i\ t\ \texttt{END} \quad \Longrightarrow \quad \texttt{AX}\ [i == t]\ \texttt{END}$$

A schema definition paragraph is formed from a box token `SCH`, a name i, a schema text t, and an `END` token. An equivalent axiomatic description paragraph is that which would be written textually as a box token `AX`, a [token, the original name i, a == token, the original schema text t, a] token, and an `END` token.

EXAMPLE 2 The semantic transformation rule for a schema hiding expression, and an informal reading of it, are as follows.

$$(e \ ;\ \mathbb{P}[\sigma]) \setminus (i_1, ..., i_n) \quad \Longrightarrow \quad \exists\ i_1 : \mathit{carrier}\ (\sigma\ i_1);\ ...;\ i_n : \mathit{carrier}\ (\sigma\ i_n) \bullet e$$

A schema with signature σ from which some names are hidden is semantically equivalent to the schema existential quantification of the hidden names from the schema. Each name is declared with the set that is the carrier set of the type of the name in the signature of the schema.

The phrase templates are patterns; they are not specific sentences and they are not written in the syntactic metalanguage. These patterns are written in a notation based on the concrete and annotated syntaxes, with metavariables appearing in place of syntactically well-formed phrases. The metavariables are defined in Tables 20 and 21 (the phrases being defined in clauses 7 and 8, and the operator words in 7.4.4). Where several phrases of the same syntactic classes have to be distinguished, these metavariables are given distinct numeric subscripts. The letters $_k$, $_m$, $_n$, $_r$ are used as metavariables for such numeric subscripts. The patterns can be viewed either as using the non-terminal symbols of the Z lexis with the -tok suffixes omitted from mathematical symbols, or as using the mathematical rendering with the box tokens in place of paragraph outlines.

Table 20 — Metavariables for phrases

Symbol	Definition
a	denotes an `ExpressionList` phrase (a for list argument).
b	denotes a list of digits within a `NUMERAL` token.
c	denotes a digit within a `NUMERAL` token.
D	denotes a `Paragraph` phrase.
d	denotes a `Declaration` phrase.
e	denotes an `Expression` phrase.
f	denotes a free type's `NAME` token.
g	denotes an injection's `NAME` token (g for in*g*ection).
h	denotes an element's `NAME` token (h for *h*element).
i, j	denote `NAME` tokens or `DeclName` or `RefName` phrases (i for identifier).
p	denotes a `Predicate` phrase.
s	denotes a `Section` phrase.
t	denotes a `SchemaText` phrase (t for text).
u, v, w, x, y	denote distinct names for new local declarations.
z	denotes a `Specification` sentence.
τ	denotes a `Type` phrase.
σ	denotes a `Signature` phrase.
$+$	denotes a `STROKE` token.
$*$	denotes a `{ STROKE }` phrase.
...	denotes elision of repetitions of surrounding phrases, the total number of repetitions depending on syntax.

The applicability of a transformation rule can be guarded by a condition written above the $\Longrightarrow$ symbol. Local definitions can be associated with a transformation rule by appending a *where* clause, in which later definitions can refer to earlier definitions.

Table 21 — Metavariables for operator words

Symbol	Definition
el	denotes an `EL` token.
elp	denotes an `ELP` token.
er	denotes an `ER` token.
ere	denotes an `ERE` token.
erep	denotes an `EREP` token.
erp	denotes an `ERP` token.
es	denotes an `ES` token.
ess	denotes an `ES` token or `SS` token.
in	denotes an `I` token.
ip	denotes an `IP` token or $\in$ token or $=$ token.
ln	denotes an `L` token.
lp	denotes an `LP` token.
post	denotes a `POST` token.
postp	denotes a `POSTP` token.
pre	denotes a `PRE` token.
prep	denotes a `PREP` token.
sr	denotes an `SR` token.
sre	denotes an `SRE` token.
srep	denotes an `SREP` token.
srp	denotes an `SRP` token.
ss	denotes an `SS` token.

The transformation rule metalanguage is used in defining characterisation rules, syntactic transformation rules, type inference rules, and semantic transformation rules (clauses 9, 12, 13 and 14).

4.4 Type inference rule metalanguage

Each type inference rule is written in the following form.

$$\frac{\textit{type subsequents}}{\textit{type sequent}}\ (\textit{side-condition})$$

where local-declaration
and ...

This can be read as: if the *type subsequents* are valid, and the *side-condition* is true, then the *type sequent* is valid, in the context of the zero-or-more *local-declarations*. The side-condition is optional; if omitted, the type inference rule is equivalent to one with a true side-condition.

The annotated syntax (see 10.2) establishes notation for writing types as `Type` phrases and for writing signatures as `Signature` phrases. The $\wr$ operator allows annotations such as types to be associated with other phrases. Determining whether a type sequent is valid or not involves manipulation of types and signatures. This requires viewing types and signatures as values, and having a mathematical notation to do the manipulation. Signatures are viewed as functions from names to type values. `Type` is used to denote the set of type values as well as the set of type phrases, the appropriate interpretation being distinguished by context of use. Similarly, `NAME` is also used to denote a set of name values. These values all lie within the type universe. A type's `NAME` has a corresponding type value in the type universe whereas a type's carrier set is in the semantic universe.

Type values are formed from just finite sets and ordered pairs, so the mathematical metalanguage introduced in 4.2 suffices for their manipulation.

Details of which names are in scope are kept in environments. The various kinds of environment are defined in

Table 22, and metavariables for environments are defined in Table 23.

Table 22 — Environments

Symbol	Definition
`TypeEnv`	denotes type environments, where `TypeEnv` == `NAME` ⇸ `Type`. Type environments associate names with types. They are like signatures, but are used in different contexts.
`SectTypeEnv`	denotes section-type environments, where `SectTypeEnv` == `NAME` ⇸ (`NAME` × `Type`). Section-type environments associate names of declarations with the name of the ancestral section that originally declared the name paired with its type.
`SectEnv`	denotes section environments, where `SectEnv` == `NAME` ⇸ `SectTypeEnv`. Section environments associate section names with section-type environments.

Table 23 — Metavariables for environments

Symbol	Definition
Σ	denotes a type environment, Σ : `TypeEnv`.
Γ	denotes a section-type environment, Γ : `SectTypeEnv`.
Λ	denotes a section environment, Λ : `SectEnv`.

Variables over the type universe are defined in Table 24, and a relation on types is defined in Table 25.

Table 24 — Variables over type universe

Symbol	Definition
α	denotes a type, α : `Type`.
β	denotes a signature, β : `Signature`, which may be a type environment.
γ	denotes a section-type environment, γ : `SectTypeEnv`.
δ	denotes a section environment, δ : `SectEnv`.

Table 25 — Type relations

Symbol	Definition
generic_type τ	asserts that τ is a generic type.

Type sequents are written using a $\vdash$ symbol superscripted with a mnemonic letter to distinguish the syntax of the phrase appearing to its right (Table 26).

NOTE 1 These superscripts are the same as the superscripts used on the $[\![\ ]\!]$ semantic brackets in the semantic relations below (Table 29).

The annotated phrases to the right of $\vdash$ in type sequents are phrase templates written using the same metavariables as the syntactic transformation rules (Table 20).

Table 26 — Type sequents

Symbol	Definition
$\vdash^{\mathcal{Z}} z$	a type sequent asserting that specification z is well-typed.
$\Lambda \vdash^{\mathcal{S}} s \mathbin{:} \Gamma$	a type sequent asserting that, in the context of section environment Λ, section s has section-type environment Γ.
$\Sigma \vdash^{\mathcal{D}} D \mathbin{:} \sigma$	a type sequent asserting that, in the context of type environment Σ, the paragraph D has signature σ.
$\Sigma \vdash^{\mathcal{P}} p$	a type sequent asserting that, in the context of type environment Σ, the predicate p is well-typed.
$\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \tau$	a type sequent asserting that, in the context of type environment Σ, the expression e has type τ.

EXAMPLE The type inference rule for a schema conjunction expression, and its informal reading, are as follows.

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{:} \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathbin{:} \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \mathbin{:} \tau_1) \land (e_2 \mathbin{:} \tau_2) \mathbin{:} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[\beta_1 \cup \beta_2] \end{array} \right)$$

In a schema conjunction expression $e_1 \land e_2$, expressions e_1 and e_2 shall be schemas, and their signatures shall be compatible. The type of the whole expression is that of the schema whose signature is the union of those of expressions e_1 and e_2.

NOTE 2 The metavariables β_1 and β_2 in this example denote syntactic phrases. These are mapped implicitly to type values, so that the set union can be computed, and the resulting signature is implicitly mapped back to a syntactic phrase. These mappings are not made explicit as they would make the type inference rules harder to read, e.g. $[\![[\![\beta_1]\!] \cup [\![\beta_2]\!]]\!]$.

This metalanguage is used in defining type inference rules (clause 13).

4.5 Semantic relation metalanguage

Most semantic relations are equations written in the following form.

$$[\![\mathit{phrase\ template}]\!] \quad = \quad \mathit{semantics}$$

Where the definition is only partial, the equality notation is not appropriate, and instead a lower bound is specified on the semantics.

$$\mathit{semantics} \quad \subseteq \quad [\![\ \mu \ e_1 \bullet e_2 \ ]\!]^{\mathcal{E}}$$

The phrase templates use the same metavariables as used by the syntactic transformation rules (Table 20).

Symbols concerned with the domains of the semantic definitions are listed in Tables 27 and 28.

The meaning of a phrase template is given by a semantic relation from the Z phrase in terms of operations of ZF set theory on the semantic universe. There are different semantic relations for each syntactic notation, written using the conventional $[\![\ ]\!]$ semantic brackets, but here superscripted with a mnemonic letter to distinguish the syntax of phrase appearing within them (Table 29).

NOTE The superscripts are the same as those used on the $\vdash$ of type sequents in the type inference rules above (Table 26).

Table 27 — Semantic universe

Symbol	Definition
$\mathbb{W}$	denotes the world of semantic values for Z expressions that are not generic, where $\mathbb{W} : \mathbb{P}\,\mathbb{U}$.
Model	denotes models, where $Model == \texttt{NAME} \pfun \mathbb{U}$. Models associate names of declarations with semantic values. They are applied only to names in their domains, as guaranteed by well-typedness.
SectionModels	denotes functions from sections' names to their sets of models, where $SectionModels == \texttt{NAME} \pfun \mathbb{P}\, Model$.

Table 28 — Variables over semantic universe

Symbol	Definition
M	denotes a model, $M : Model$.
T	denotes a section's name and its set of models, $T : SectionModels$.
t	denotes a binding semantic value, $t : \texttt{NAME} \pfun \mathbb{W}$.
u	denotes a generic semantic value, $u : \mathbb{U} \setminus \mathbb{W}$.
w, x, y	denote non-generic semantic values, $w : \mathbb{W}$; $x : \mathbb{W}$; $y : \mathbb{W}$.

Table 29 — Semantic relations

Symbol	Definition
$[\![z]\!]^{\mathcal{Z}}$	denotes the meaning of specification z, where $[\![z]\!]^{\mathcal{Z}} \in SectionModels$. The meaning of a specification is the function from its sections' names to their sets of models.
$[\![s]\!]^{\mathcal{S}}$	denotes the meaning of section s, where $[\![s]\!]^{\mathcal{S}} \in SectionModels \pfun SectionModels$. The meaning of a section is given by the extension of a *SectionModels* function with an extra maplet corresponding to the given section.
$[\![D]\!]^{\mathcal{D}}$	denotes the meaning of paragraph D, where $[\![D]\!]^{\mathcal{D}} \in Model \rel Model$. The meaning of a paragraph relates a model to that model extended according to that paragraph.
$[\![p]\!]^{\mathcal{P}}$	denotes the meaning of predicate p, where $[\![p]\!]^{\mathcal{P}} \in \mathbb{P}\, Model$. The meaning of a predicate is the set of all models in which that predicate is true.
$[\![e]\!]^{\mathcal{E}}$	denotes the meaning of expression e, where $[\![e]\!]^{\mathcal{E}} \in Model \fun \mathbb{W}$. The meaning of an expression is a function returning the semantic value of the expression in the given model.
$[\![\tau]\!]^{\mathcal{T}}$	denotes the meaning of type τ, where $[\![\tau]\!]^{\mathcal{T}} \in Model \pfun \mathbb{P}\,\mathbb{U}$. The meaning of a type is the semantic value of its carrier set, as determined from the given model.

EXAMPLE The semantic relation for a conjunction predicate, and its informal reading, are as follows. The conjunction predicate $p_1 \land p_2$ is *true* if and only if p_1 and p_2 are *true*.

$$[\![p_1 \land p_2]\!]^{\mathcal{P}} = [\![p_1]\!]^{\mathcal{P}} \cap [\![p_2]\!]^{\mathcal{P}}$$

In terms of the semantic universe, it is *true* in those models in which both p_1 and p_2 are *true*, and is *false* otherwise.

Within the semantic relations, the idioms listed in Table 30 occur frequently.

Table 30 — Semantic idioms

Idiom	Description
$[\![e]\!]^{\mathcal{E}} M$	denotes the value of expression e in model M
$M \oplus t$	denotes the model M giving semantic values for more global declarations overridden by the binding t giving the semantic values of locally declared names

Semantic relation metalanguage is used in defining semantic relations (clause 15).

5 Conformance

5.1 Phases of the definition

The definition of the Z notation is divided into a sequence of phases, as illustrated in Figure 1. Each arrow represents a phase from a representation of a Z specification at its source to another representation of the Z specification at its target. The phase is named at the left margin. Some phases detect errors in the specification; these are shown drawn off to the right-hand side.

NOTE 1 Figure 1 shows the order in which the phases are applied, and where errors are detected; it does not show information flows.

NOTE 2 The arrows are analogous to total and partial function arrows in the Z mathematical toolkit, but drawn vertically.

5.2 Conformance requirements

5.2.1 Specification conformance

For a Z specification to conform to this International Standard, no errors shall be detected by any of the phases shown in Figure 1. In words, for a Z specification to conform to this International Standard, its formal text shall be valid mark-up of a sequence of Z characters, that can be lexed as a valid sequence of tokens, that can be parsed as a sentence of the concrete syntax, and that is well-typed according to the type inference system.

NOTE The presence of sections that have no models does not affect the conformance of their specification.

5.2.2 Mark-up conformance

A mark-up for Z based on LaTeX [10] conforms to this International Standard if and only if it follows the rules given for LaTeX mark-up in A.2.

A mark-up for Z used in e-mail correspondence conforms to this International Standard if and only if it follows the rules given for e-mail mark-up in A.3.

Mark-up for Z based on any other mark-up language is permitted; it shall be possible to define a functional mapping from that mark-up to sequences of Z characters.

The ISO/IEC 10646 [5][6] representation may be used directly.

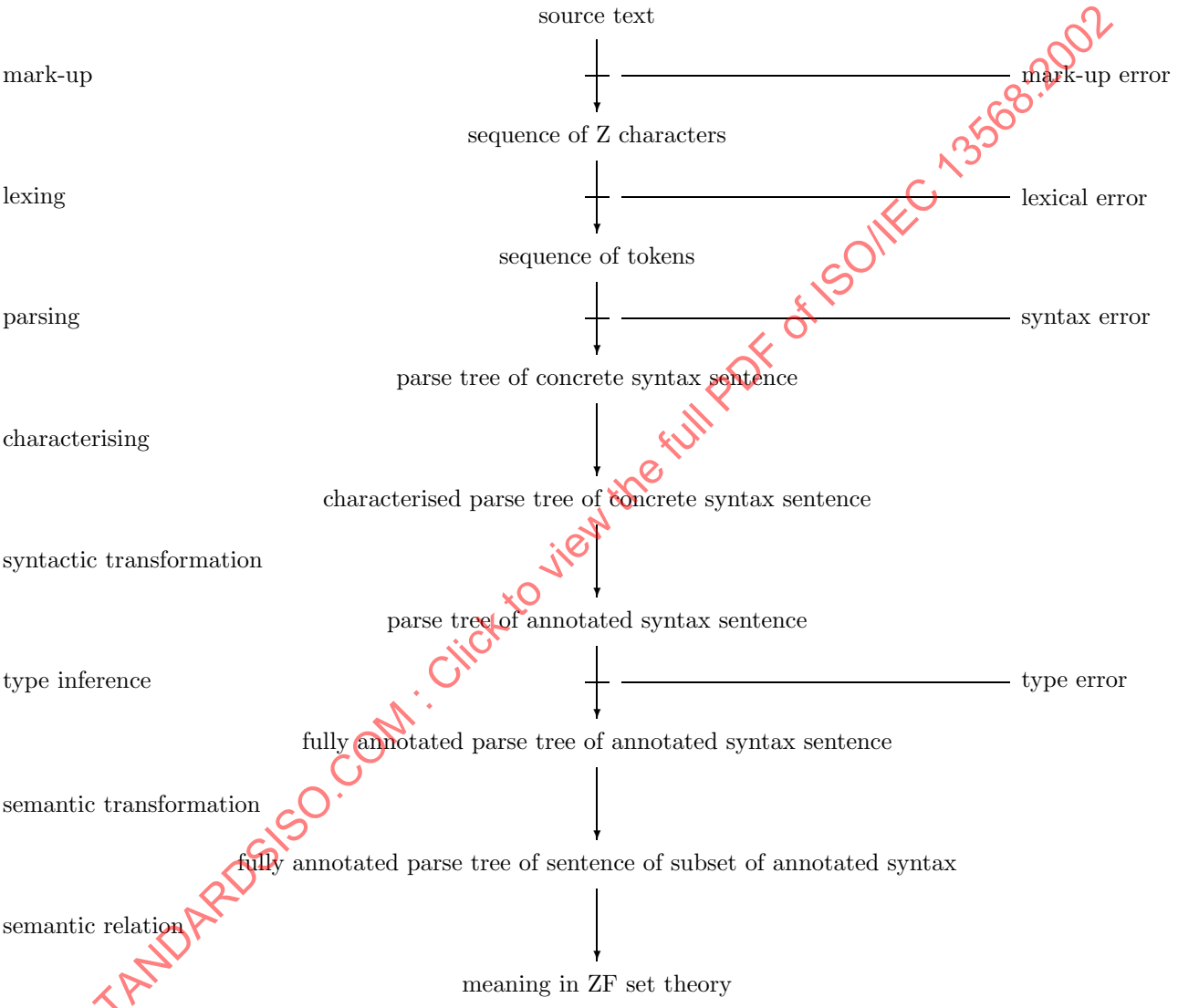


Figure 1 — Phases of the definition

5.2.3 Deductive system conformance

A Z deductive system conforms to this International Standard if and only if its rules are sound with respect to the semantics, i.e. if both of the following conditions hold:

a) all of its axioms hold in all models of all Z specifications, i.e. for any axiom p,

$$[\![p]\!]^{\mathcal{P}} = Model$$

b) all of its rules of inference have the property that the intersection of the sets of models of each of the premises is contained in the model of the conclusion, i.e. for any rule of inference where p is deduced from p_1, ... p_n,

$$[\![p_1]\!]^{\mathcal{P}} \cap ... \cap [\![p_n]\!]^{\mathcal{P}} \subseteq [\![p]\!]^{\mathcal{P}}$$

All constraints appearing before a conjecture in a specification may be used as premises in inferences about that conjecture. The documentation of a Z deductive system should clarify whether or not it allows constraints appearing after a conjecture in a specification to be used as premises in inferences about that conjecture.

The semantic relations assigning meanings to Z phrases are defined loosely by this International Standard, so that there is a set of possible conforming semantic relations. A deductive system conforms if and only if its rules are sound with respect to one or more of these semantic relations. The documentation of a Z deductive system should indicate the semantic relations to which its rules are sound.

5.2.4 Mathematical toolkit conformance

A Z section whose name is the same as a section of the mathematical toolkit of annex B conforms to this International Standard if and only if it defines the same set of models as that section of the mathematical toolkit. A Z section whose name is *prelude* conforms to this International Standard if and only if it defines the same set of models as the section in clause 11.

A mathematical toolkit conforms to this standard if it defines a conformant section called standard_toolkit.

NOTE 1 The set of models defined by a section within a specification may be found by applying the meaning of the specification to the section's name.

NOTE 2 A conforming section of the toolkit may formulate its definitions differently from those in annex B.

NOTE 3 A conforming section of the toolkit may partition its definitions amongst parent sections that differ from those in annex B.

NOTE 4 Alternative and additional toolkits are not precluded, but are required to have different section names to avoid confusion.

NOTE 5 Some names are loosely defined, such as $\mathbb{A}$, and may be further constrained by sections that use toolkit sections, but not by toolkit sections themselves.

5.2.5 Support tool conformance

A strongly conforming Z support tool shall recognise at least one conforming mark-up, accepting all conforming Z specifications presented to it, and rejecting all non-conforming Z specifications presented to it. A weakly conforming Z support tool shall never accept a non-conforming Z specification, nor reject a conforming Z specification, but it may state that it is unable to determine whether or not a Z specification conforms.

NOTE Strong conformance can be summarised as always being right, whereas weak conformance is never being wrong.

EXAMPLE A tool would be weakly conformant if it were to announce its inability to determine the conformance of a Z specification that used names longer than the tool could handle, but would be non-conformant if it silently truncated long names.

Certain exceptions to general rules are anticipated and permitted in subsequent clauses, because of, for example, implementation considerations or for backwards compatibility with pre-existing tools.

5.3 Structure of this document

The phases in the definition of the Z notation, and the representations of specifications manipulated by those phases, as illustrated in Figure 1, are specified in the following clauses and annexes.

Annex A, Mark-ups, specifies two source text representations and corresponding mark-up phases for translating source text to sequences of Z characters.

Clause 6 specifies the Z characters by their appearances and their names in ISO/IEC 10646.

Clause 7, Lexis, specifies tokens and the lexing phase that translates a sequence of Z characters to a sequence of tokens.

Clause 8 specifies the grammar of the concrete syntax, and hence abstractly specifies the parsing phase that translates a sequence of tokens to a parse tree of a concrete syntax sentence.

Clause 9 specifies the characterising phase, during which characteristic tuples are made explicit in the parse tree of a concrete syntax sentence.

Clause 10 specifies the grammar of the annotated syntax, defining the target language for the syntactic transformation phase.

Clause 11 specifies the prelude section, providing the initial environment of definitions.

Clause 12 specifies the syntactic transformation phase that translates a parse tree of a concrete syntax sentence to a parse tree of an equivalent annotated syntax sentence.

Clause 13 specifies the type inference phase, during which type annotations are added to the parse tree of the annotated syntax sentence, and reference expressions that refer to generic definitions are translated to generic instantiation expressions.

Clause 14 specifies the semantic transformation phase, during which some annotated parse trees are translated to equivalent other annotated parse trees.

Clause 15 specifies the semantic relation between a sentence of the remaining annotated syntax and its meaning in ZF set theory.

Annex C duplicates those parts of the definition that fit into an organisation by concrete syntax production.

6 Z characters

6.1 Introduction

A Z character is the smallest unit of information in Z. Z characters are used to build tokens (clause 7), which are in turn the units of information in the concrete syntax (clause 8). The Z characters are defined by reference to ISO/IEC 10646-1 [5] and ISO/IEC 10646-2 [6]: the appearance, code position and name of each Z character are listed.

Many Z characters are not present in the standard 7-bit ASCII encoding [4]. It is possible to represent Z characters in ASCII, by defining a mark-up, where several ASCII characters are used together to represent a single Z character. This International Standard defines some ASCII mark-ups in annex A by relation to the ISO/IEC 10646 representation (henceforth called UCS) defined here. Other mark-ups of Z characters can similarly be defined by relation to the UCS representation.

6.2 Formal definition of Z characters


```
ZCHAR          = DIGIT | LETTER | SPECIAL | SYMBOL ;

DIGIT          = DECIMAL
               | ? other UCS chars with Number property but Number, Decimal Digit (as supported) ?
               ;

DECIMAL        = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
               | ? any other UCS characters with Number, Decimal Digit property (as supported) ?
               ;

LETTER         = LATIN | GREEK | OTHERLETTER
               | ? any characters of the mathematical toolkit with letter property (as supported) ?
               | ? any other UCS characters with letter property (as supported) ?
               ;

LATIN          = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I'
               | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
               | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
               | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
               | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r'
               | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
               ;

GREEK          = 'Δ' | 'Ξ' | 'θ' | 'λ' | 'μ' ;

OTHERLETTER    = '𝔸' | 'ℕ' | 'ℙ' ;

SPECIAL        = STROKECHAR | WORDGLUE | BRACKET | BOXCHAR | NLCHAR | SPACE ;

STROKECHAR     = '′' | '!' | '?' ;

WORDGLUE       = '↗' | '↙' | '↘' | '↖' | '_' ;

BRACKET        = '(' | ')' | '[' | ']' | '{' | '}' | '⦇' | '⦈' | '⟪' | '⟫' ;

BOXCHAR        = ZEDCHAR | AXCHAR | SCHCHAR | GENCHAR | ENDCHAR ;

SYMBOL         = '|' | '&' | '⊢' | '∧' | '∨' | '⇒' | '⇔' | '¬' | '∀' | '∃'
               | '×' | '/' | '=' | '∈' | ':' | ';' | ',' | '.' | '•'
               | '\' | '⨾' | '⨟' | '≫' | '+'
               | ? any characters of the mathematical toolkit with neither letter or
                   number property (as supported) ?
               | ? any other UCS characters with neither letter or
                   number property and that are not in SPECIAL (as supported) ?
               ;
```

6.3 Additional restrictions and notes

The word *supported* means available for use in presenting a specification.

The characters enumerated in the formal definition are those used by the Z core language; they shall be supported. If the mathematical toolkit is supported, then its characters shall be supported. The "other UCS characters" may also be supported, extending `DIGIT`, `DECIMAL`, `LETTER` or `SYMBOL` according to their property, but not extending `SPECIAL`. Use of characters that are absent from UCS is permitted, but there is no standard way of distinguishing which of `DIGIT`, `DECIMAL`, `LETTER` or `SYMBOL` (not `SPECIAL`) they extend, and specifications using them might not be interchangeable between tools.

NOTE 1 `SPACE` is a Z character that serves to separate two sequences of Z characters that would otherwise be mis-lexed as a single token.

NOTE 2 `BOXCHAR` characters correspond to Z's distinctive boxes around paragraphs. The `NLCHAR` character is used to mark a hard newline (see 7.5).

6.4 Z character representations

6.4.1 Introduction

The following tables show the Z characters in their mathematical representation. (Other representations are given in annex A.) The columns give:

Math: The representation for rendering the character on a high resolution device, such as a bit-mapped screen, or on paper (either hand-written, or printed).

Code position: The encoding of the Z character in UCS.

Character name: The name for the character in UCS.

NOTE The code position column is included to assist location of the characters by users; it is not necessary for definition of the characters.

6.4.2 Decimal digit characters

Math	Code position	Character name
0	0000 0030	DIGIT ZERO
⋮	⋮	⋮
9	0000 0039	DIGIT NINE

For each `DECIMAL` character, UCS defines a corresponding *decimal digit value*. This is used in 7.3.

6.4.3 Letter characters

6.4.3.1 Latin alphabet characters

Math	Code position	Character name
A	0000 0041	LATIN CAPITAL LETTER A
⋮	⋮	⋮
Z	0000 005A	LATIN CAPITAL LETTER Z
a	0000 0061	LATIN SMALL LETTER A
⋮	⋮	⋮
z	0000 007A	LATIN SMALL LETTER Z

6.4.3.2 Greek alphabet characters

The Greek alphabet characters used by the Z core language are those listed here.

Math	Code position	Character name
Δ	0000 0394	GREEK CAPITAL LETTER DELTA
Ξ	0000 039E	GREEK CAPITAL LETTER XI
θ	0000 03B8	GREEK SMALL LETTER THETA
λ	0000 03BB	GREEK SMALL LETTER LAMBDA
μ	0000 03BC	GREEK SMALL LETTER MU

6.4.3.3 Other Z core language letter characters

The other Z core language characters with UCS *letter* property are those of the prelude (clause 11), as listed here.

Math	Code position	Character name
$\mathbb{A}$	0001 D538	MATHEMATICAL DOUBLE-STRUCK CAPITAL A
$\mathbb{N}$	0000 2115	DOUBLE-STRUCK CAPITAL N
$\mathbb{P}$	0000 2119	DOUBLE-STRUCK CAPITAL P

6.4.4 Special characters

6.4.4.1 Stroke characters

Math	Code position	Character name
$'$	0000 02B9	MODIFIER LETTER PRIME
!	0000 0021	EXCLAMATION MARK
?	0000 003F	QUESTION MARK

6.4.4.2 Word glue characters

The characters '↗', '↙', '↘', and '↖' may be presented as in-line literals, or they may indicate a raising/lowering of the text, and possible size change. Such rendering details are not defined here.

Math	Code position	Character name
$\nearrow$	0000 2197	NORTH EAST ARROW
$\swarrow$	0000 2199	SOUTH WEST ARROW
$\searrow$	0000 2198	SOUTH EAST ARROW
$\nwarrow$	0000 2196	NORTH WEST ARROW
_	0000 005F	LOW LINE

6.4.4.3 Bracket characters

Math	Code position	Character name
(	0000 0028	LEFT PARENTHESIS
)	0000 0029	RIGHT PARENTHESIS
[	0000 005B	LEFT SQUARE BRACKET
]	0000 005D	RIGHT SQUARE BRACKET
{	0000 007B	LEFT CURLY BRACKET
}	0000 007D	RIGHT CURLY BRACKET
$\langle\!\vert$	0000 2989	Z NOTATION LEFT BINDING BRACKET
$\vert\!\rangle$	0000 298A	Z NOTATION RIGHT BINDING BRACKET
$\langle\langle$	0000 300A	LEFT DOUBLE ANGLE BRACKET
$\rangle\rangle$	0000 300B	RIGHT DOUBLE ANGLE BRACKET

6.4.4.4 Box characters

Box characters are assembled to form box tokens (see 7.2), which in turn correspond to boxes around paragraphs (see 8.5). The simple renderings of the characters are suggestive of the boxes around paragraphs. The `ENDCHAR` character is used to mark the end of a `Paragraph`.

NOTE These box characters are not intended to be used for rendering the boxes around paragraphs.

Z character	Simple rendering	Code position	Character name
`ZEDCHAR`	|	`0000 2028`	LINE SEPARATOR
`AXCHAR`	|	`0000 2577`	BOX DRAWINGS LIGHT DOWN
`SCHCHAR`	⌜	`0000 250C`	BOX DRAWINGS LIGHT DOWN AND RIGHT
`GENCHAR`	=	`0000 2550`	BOX DRAWINGS DOUBLE HORIZONTAL
`ENDCHAR`	(new line)	`0000 2029`	PARAGRAPH SEPARATOR

6.4.4.5 Other `SPECIAL` characters

Z character	Simple rendering	Code position	Character name
`NLCHAR`	(new line)	`0000 000A`	LINE FEED
`SPACE`	(space)	`0000 0020`	SPACE

6.4.5 Symbol characters except mathematical toolkit characters

Math	Code position	Character name
|	`0000 007C`	VERTICAL LINE
&	`0000 0026`	AMPERSAND
⊢	`0000 22A2`	RIGHT TACK
∧	`0000 2227`	LOGICAL AND
∨	`0000 2228`	LOGICAL OR
⇒	`0000 21D2`	RIGHTWARDS DOUBLE ARROW
⇔	`0000 21D4`	LEFT RIGHT DOUBLE ARROW
¬	`0000 00AC`	NOT SIGN
∀	`0000 2200`	FOR ALL
∃	`0000 2203`	THERE EXISTS
×	`0000 00D7`	MULTIPLICATION SIGN
/	`0000 002F`	SOLIDUS
=	`0000 003D`	EQUALS SIGN
∈	`0000 2208`	ELEMENT OF
:	`0000 003A`	COLON
;	`0000 003B`	SEMICOLON
,	`0000 002C`	COMMA
.	`0000 002E`	FULL STOP
•	`0000 2981`	Z NOTATION SPOT
\	`0000 29F9`	BIG REVERSE SOLIDUS
↾	`0000 2A21`	Z NOTATION SCHEMA PROJECTION
⨾	`0000 2A1F`	Z NOTATION SCHEMA COMPOSITION
≫	`0000 2A20`	Z NOTATION SCHEMA PIPING
+	`0000 002B`	PLUS SIGN

6.4.6 Mathematical toolkit characters

The mathematical toolkit (annex B) need not be supported by an implementation. If it is supported, it shall use the representations given here.

Mathematical toolkit names that use only Z core language characters, or combinations of Z characters defined here, are not themselves listed here.

6.4.6.1 Section set_toolkit

Math	Code position	Character name
↔	0000 2194	LEFT RIGHT ARROW
→	0000 2192	RIGHTWARDS ARROW
≠	0000 2260	NOT EQUAL TO
∉	0000 2209	NOT AN ELEMENT OF
∅	0000 2205	EMPTY SET
⊆	0000 2286	SUBSET OF OR EQUAL TO
⊂	0000 2282	SUBSET OF
∪	0000 222A	UNION
∩	0000 2229	INTERSECTION
\	0000 005C	REVERSE SOLIDUS
⊖	0000 2296	CIRCLED MINUS
⋃	0000 22C3	N-ARY UNION
⋂	0000 22C2	N-ARY INTERSECTION
𝔽	0001 D53D	MATHEMATICAL DOUBLE-STRUCK CAPITAL F

6.4.6.2 Section relation_toolkit

Math	Code position	Character name
↦	0000 21A6	RIGHTWARDS ARROW FROM BAR
⨾	0000 2A3E	Z NOTATION RELATIONAL COMPOSITION
∘	0000 2218	RING OPERATOR
◁	0000 25C1	WHITE LEFT-POINTING TRIANGLE
▷	0000 25B7	WHITE RIGHT-POINTING TRIANGLE
⩤	0000 2A64	Z NOTATION DOMAIN ANTIRESTRICTION
⩥	0000 2A65	Z NOTATION RANGE ANTIRESTRICTION
~	0000 223C	TILDE OPERATOR
⦇	0000 2987	Z NOTATION LEFT IMAGE BRACKET
⦈	0000 2988	Z NOTATION RIGHT IMAGE BRACKET
⊕	0000 2295	CIRCLED PLUS

6.4.6.3 Section function_toolkit

Math	Code position	Character name
⇸	0000 21F8	RIGHTWARDS ARROW WITH VERTICAL STROKE
⤔	0000 2914	RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
↣	0000 21A3	RIGHTWARDS ARROW WITH TAIL
⤀	0000 2900	RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
↠	0000 21A0	RIGHTWARDS TWO-HEADED ARROW
⤖	0000 2916	RIGHTWARDS TWO-HEADED ARROW WITH TAIL
⇻	0000 21FB	RIGHTWARDS ARROW WITH DOUBLE VERTICAL STROKE
⤕	0000 2915	RIGHTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE

6.4.6.4 Section number_toolkit

Math	Code position	Character name
$\mathbb{Z}$	0000 2124	DOUBLE-STRUCK CAPITAL Z
-	0000 002D	HYPHEN-MINUS
$-$	0000 2212	MINUS SIGN
$\leq$	0000 2264	LESS-THAN OR EQUAL TO
$<$	0000 003C	LESS-THAN SIGN
$\geq$	0000 2265	GREATER-THAN OR EQUAL TO
$>$	0000 003E	GREATER-THAN SIGN
$*$	0000 002A	ASTERISK

6.4.6.5 Section sequence_toolkit

Math	Code position	Character name
#	0000 0023	NUMBER SIGN
$\langle$	0000 3008	LEFT ANGLE BRACKET
$\rangle$	0000 3009	RIGHT ANGLE BRACKET
$\frown$	0000 2040	CHARACTER TIE
$\upharpoonleft$	0000 21BF	UPWARDS HARPOON WITH BARB LEFTWARDS
$\upharpoonright$	0000 21BE	UPWARDS HARPOON WITH BARB RIGHTWARDS

6.4.7 Renderings of Z characters

Renderings of Z characters are called glyphs (following the terminology of UCS). The glyphs used for Z characters are device-dependent: a rendering of a Z character on a screen is typically different from its rendering on a piece of paper.

A Z character may also be rendered using different glyphs at different places in a specification, for reasons of emphasis or aesthetics, but such different glyphs all represent the same Z character. For example, '*d*', 'd', '`d`' and '**d**' are all the same Z character. For historical reasons, the following similar-looking glyphs represent different Z characters.

- Schema composition '$\fatsemi$' and the mathematical toolkit character relational composition '$\fatsemi$' are different Z characters.
- Schema projection '$\upharpoonright$' and the mathematical toolkit character filter '$\upharpoonright$' are different Z characters.
- Schema hiding '$\backslash$' and the mathematical toolkit character set minus '$\backslash$' are different Z characters.

7 Lexis

7.1 Introduction

The lexis specifies a function from sequences of Z characters to sequences of tokens. The domain of the function involves all the Z characters of clause 6. The range of the function involves all the tokens used in clause 8. The function is partial: sequences of Z characters that do not conform to the lexis are excluded from consideration at this stage. In each association of a sequence of Z characters with a token, the sequence of Z characters is called the spelling of the token.

The lexis is composed of two parts: a context-free part followed by a context-sensitive part. The former translates the stream of Z characters into a stream of `DECORWORD`s and tokens. The latter classifies each `DECORWORD` as being a keyword, an operator token, or a `NAME` token, taking into account its spelling and the lexical scopes of the operators (see 7.4.4).

7.2 Formal definition of context-free lexis


```
TOKENSTREAM     = { SPACE } , { TOKEN , { SPACE } } ;

TOKEN           = DECORWORD | NUMERAL | STROKE
                | (-tok | )-tok | [-tok | ]-tok | {-tok | }-tok | ⟮ | ⟯ | ⟪ | ⟫
                | ZED | AX | GENAX | SCH | GENSCH | END | NL
                ;

DECORWORD       = WORD , { STROKE } ;

WORD            = WORDPART , { WORDPART }
                | ( LETTER | ( DIGIT — DECIMAL ) ) , ALPHASTR , { WORDPART }
                | SYMBOL , SYMBOLSTR , { WORDPART }
                ;

WORDPART        = WORDGLUE , ( ALPHASTR | SYMBOLSTR ) ;
ALPHASTR        = { LETTER | DIGIT } ;
SYMBOLSTR       = { SYMBOL } ;

NUMERAL         = NUMERAL , DECIMAL
                | DECIMAL
                ;

STROKE          = STROKECHAR
                | '↘' , DECIMAL , '↖'
                ;

(-tok           = '(' ;
)-tok           = ')' ;
[-tok           = '[' ;
]-tok           = ']' ;
{-tok           = '{' ;
}-tok           = '}' ;
⟮               = '⟮' ;
⟯               = '⟯' ;
⟪               = '⟪' ;
⟫               = '⟫' ;

ZED             = ZEDCHAR ;
AX              = AXCHAR ;
SCH             = SCHCHAR ;
GENAX           = AXCHAR , GENCHAR ;
GENSCH          = SCHCHAR , GENCHAR ;
END             = ENDCHAR ;
NL              = NLCHAR ;
```

7.3 Additional lexical restrictions, notes and examples

Words are formed from alphanumeric and symbolic parts.

EXAMPLE 1 The following strings of Z characters are single DECORWORDs: '&+=', 'x_+_y', 'x_+y', 'x^+y', '$x^+{}_+y$'. However, '$x+y$' comprises the three DECORWORDs 'x', '+' and 'y'.

EXAMPLE 2 The following strings of Z characters are single DECORWORDs: 'λS', 'ΔS', '$\exists\times$', '$\exists\_X$', '$\exists_X$'. However, '$\exists X$' is the keyword token '$\exists$', followed by the DECORWORD 'X'.

EXAMPLE 3 The following strings of Z characters are single DECORWORDs: '$\times:\in$', '$x\_:\_e$', '$_x:_e$'. However, '$x:e$' is the word 'x', followed by the keyword token ':', followed by the DECORWORD 'e'.

SPACE is not itself lexed as part of any token. It can be used freely between tokens. The cases where its use is necessary are: between two WORDs, which would otherwise be lexed as a single WORD; between an alphabetic WORD and a NUMERAL, which would otherwise be lexed as a single WORD; between a DECORWORD and a STROKE, which would otherwise be lexed as a single DECORWORD; and between two consecutive NUMERALs, which would otherwise be lexed as a single NUMERAL.

EXAMPLE 4 abc is the DECORWORD 'abc'.
$a\ bc$ is the DECORWORD 'a' followed by the DECORWORD 'bc'.
$a12$ is the DECORWORD '$a12$'.
$a\ 12$ is the DECORWORD 'a' followed by the NUMERAL '12'.
$x!$ is the DECORWORD comprising the WORD 'x' followed by the STROKE '!'.
$x\ !$ is the DECORWORD 'x' followed by the STROKE '!'.
$x!\ !$ is the DECORWORD '$x!$' followed by the STROKE '!'.

The lexis allows a WORD to include subscript digits; it also allows a DECORWORD to be decorated with subscript decimal digits. Trailing subscript decimal digits shall be lexed as strokes, not as part of a WORD.

EXAMPLE 5 x_{a1} is a DECORWORD comprising the WORD 'x_a' and the STROKE '$_1$'.
$x_a?$ is a DECORWORD comprising the WORD 'x_a' and the STROKE '?'.
$x_{1\,a}$ is a DECORWORD comprising the WORD '$x_{1\,a}$' and no strokes.
a^{b_3} is a DECORWORD comprising the WORD 'a^{b_3}' and no strokes.

A multi-digit last WORDPART enclosed in a $\searrow\ldots\nwarrow$ pair is deprecated, because of the visual ambiguity with multiple STROKE subscript digits.

Throughout subsequent phases of this International Standard, NUMERALs are considered to comprise only the decimal digits enumerated in 6.4.2. Any other decimal digit in a NUMERAL is transformed to the enumerated decimal digit that has the same decimal digit value as defined by UCS.

EXAMPLE MATHEMATICAL DOUBLE-STRUCK DIGIT FIVE, code position 0001 D7DD, has decimal digit value 5 and so, whenever it appears in a NUMERAL, is transformed to DIGIT FIVE, code position 0000 0035.

NOTE 1 Although a parser does not need to know the spelling of particular instances of DECORWORD, NUMERAL, STROKE, etc tokens, subsequent phases of processing do. For example, references to undeclared names are excluded by type inference rules, and the values of numerals may be determined from the values of their decimal digits. The relation between instances of tokens and spellings is not explicitly formalized here. The relation between instances of NUMERALs and their values is formalized for those used as number literal expressions (see 12.2.6.9), because the semantic values of number literal expressions appear in models. The relation between other instances of NUMERALs (those used in tuple selection expressions and those used as operator precedences) and their values is not formalised; these values accord with the usual decimal convention.

Tools may impose restrictions on the forms of names. Section names that are entirely ASCII, alphanumeric, capitalized and short are the most likely to be portable between tools.

7.4 Context-sensitive lexis

7.4.1 Introduction

The context-sensitive part of lexis maps each DECORWORD to either a keyword token, an operator token, or a NAME token. It also strips all SPACEs from the token stream, and forwards all other tokens unchanged.

If a DECORWORD's spelling is exactly that of a keyword, the DECORWORD is mapped to the corresponding keyword token. Otherwise, if the DECORWORD's WORD part's spelling is that of an operator word, the DECORWORD is mapped to the relevant operator token. Otherwise, the DECORWORD is mapped to a NAME token.

In the case of a NAME token, for every '↘' WORDGLUE character in its WORD part, there shall be a paired following '↖' WORDGLUE character, for every '↗' WORDGLUE character in its WORD part, there shall be a paired following '↙' WORDGLUE character, and these shall occur only in nested pairs.

NOTE 1 Operators have a similar restriction applied to the whole operator name (see 12.2.8), not to the individual words within the operator.

The keywords are as listed in the following tables. No other spellings give rise to keyword tokens. The columns give:

Spelling: The sequence of Z characters representing the rendering of the token on a high resolution device, such as a bit-mapped screen, or on paper (either hand-written, or printed).

Token: The token used for that keyword in the concrete syntax.

Token name: A suggested form for reading the keyword out loud, suitable for use in reviews, or for discussing specifications over the telephone. In the following, an English language form is given; for other natural languages, other forms may be defined.

NOTE 2 Even where a keyword consists of a single Z character, the token name tends to reflect the keyword's function rather than the form of the Z character.

7.4.2 Alphabetic keywords

Spelling	Token	Token name
else	else	else
false	false	false
function	function	function
generic	generic	generic
if	if	if
leftassoc	leftassoc	left [associative]
let	let	let
ℙ	ℙ	powerset
parents	parents	parents
pre	pre	pre[condition]
relation	relation	relation
rightassoc	rightassoc	right [associative]
section	section	section
then	then	then
true	true	true

7.4.3 Symbolic keywords

Spelling	Token	Token name
:	:	colon
==	==	define equal
,	,-tok	comma
::=	::=	free equals
|	|-tok	bar
&	&	and also [free types]
\	\	hide
/	/	rename
.	.	select | dot
;	; -tok	semi[colon]
_	_	arg[ument]
, ,	, ,	list arg[ument]
=	=-tok	equals

EXAMPLE 1 = is recognised as the keyword token =-tok; := is recognised as a `NAME` token; ::= is recognised as the keyword token.

Spelling	Token	Token name
$\vdash?$	$\vdash?$	conjecture
$\forall$	$\forall$	for all
$\bullet$	$\bullet$	spot
$\exists$	$\exists$	exists
$\exists_1$	$\exists_1$	unique exists
$\Leftrightarrow$	$\Leftrightarrow$	equivalent | if and only if
$\Rightarrow$	$\Rightarrow$	implies
$\vee$	$\vee$	or
$\wedge$	$\wedge$	and
$\neg$	$\neg$	not
$\in$	$\in$	in | member of | element of
$\upharpoonright$	$\upharpoonright$	project
$\times$	$\times$	cross
λ	λ	lambda
μ	μ	mu
θ	θ	theta
$\overset{\circ}{9}$	$\overset{\circ}{9}$	schema compose
$\gg$	$\gg$	schema pipe

EXAMPLE 2 $\exists$ and $\exists_1$ (‘$\exists$’, ‘$\searrow$’, ‘1’, ‘$\nwarrow$’) are recognised as keyword tokens; $\exists_0$ (‘$\exists$’, ‘$\searrow$’, ‘0’, ‘$\nwarrow$’) is recognised as a `NAME` token.

EXAMPLE 3 λ is recognised as the keyword token; λx is recognised as a `NAME` token; $\lambda\ x$ is recognised as the keyword token followed by a `NAME` token.

7.4.4 User-defined operators

Each operator template creates additional keyword-like associations between `WORD`s and operator tokens. The scope of these associations is the whole of the section in which the operator template appears (not just from the operator template onwards), as well as all sections of which that section is an ancestor, excluding section headers.

NOTE The set of active associations is always a function. This International Standard does not specify how that function is determined: operator template paragraphs provide the information, yet in their concrete syntax it is assumed that the function is already known.

The appropriate token for an operator word is as follows.

PREP	prefix unary relation
PRE	prefix unary function or generic
POSTP	postfix unary relation
POST	postfix unary function or generic
IP	infix binary relation
I	infix binary function or generic
LP	left bracket of non-unary relation
L	left bracket of non-unary function or generic
ELP	first word preceded by expression of non-unary relation
EL	first word preceded by expression of non-unary function or generic
ERP	right bracket preceded by expression of non-unary relation
ER	right bracket preceded by expression of non-unary function or generic
SRP	right bracket preceded by list argument of non-unary relation
SR	right bracket preceded by list argument of non-unary function or generic
EREP	last word followed by expression and preceded by expression of tertiary or higher relation
ERE	last word followed by expression and preceded by expression of tertiary or higher function or generic
SREP	last word followed by expression and preceded by list argument of tertiary or higher relation
SRE	last word followed by expression and preceded by list argument of tertiary or higher function or generic
ES	middle word preceded by expression of non-unary operator
SS	middle word preceded by list argument of non-unary operator

EXAMPLE 1 The operator template paragraph for the (_ + _) operator adds one entry to the mapping.

Spelling	**Token**
+	I

EXAMPLE 2 The operator template paragraph for the (_ (| _ |)) operator adds two entries to the mapping.

Spelling	**Token**
(|	EL
|)	ER

EXAMPLE 3 The operator template paragraph for the (*disjoint* _) operator adds one entry to the mapping.

Spelling	**Token**
disjoint	PREP

EXAMPLE 4 The operator template paragraph for the (⟨ _ ⟩) operator adds two entries to the mapping.

Spelling	**Token**
⟨	L
⟩	SR

7.5 Newlines

The Z character NLCHAR is lexed either as a token separator (like the SPACE character) or as the token NL, depending on its context. A *soft newline* is a NLCHAR that is lexed as a token separator. A *hard newline* is a NLCHAR that is lexed as a NL token.

Tokens are assigned to a *newline category*, namely BOTH, AFTER, BEFORE or NEITHER, based on whether that token could start or end a Z phrase.

- BOTH: newlines are soft before and after the token, because it is infix, something else has to appear before it and after it.

  else function generic leftassoc parents relation rightassoc section then
  ::= |-tok ⦇ ⦈ & ⊢? , , ∧ ∨ ⇒ ⇔ × / =-tok ∈ == : ; -tok ,-tok . • ○ ⨾ ⨟ ≫
  I IP EL ELP ERE EREP ES SS SRE SREP

  All newlines are soft outside of a DeclPart or a Predicate.

  NOTE Tokens that cannot appear in these contexts are in category BOTH. This includes the box tokens. Newlines at the very beginning or end of a specification are soft.

- AFTER: newlines are soft after the token, because it is prefix, something else has to appear after it.

  if let pre
  [−tok _ ¬ ∀ ∃ $\exists_1$ ℙ (−tok {−tok ⦇ λ μ θ
  PRE PREP L LP

- BEFORE: newlines are soft before the token, because it is postfix, something else has to appear before it.

  ]−tok)−tok }−tok ⦈
  POST POSTP ER ERP SR SRP

- NEITHER: no newlines are soft, because such a token is nofix, nothing else need appear before or after it.

  false true
  NAME NUMERAL STROKE

For each NLCHAR, the newline categories of the closest token generated from the preceding Z characters and the token generated from the immediately following Z characters are examined. If either token allows the newline to be soft in that position, it is soft, otherwise it is hard (and hence recognised as a NL token).

The operator template paragraph allows the definition of various mixfix names (see 7.4.4), which are placed in the appropriate newline category. Other (ordinary) user declared names are nofix, and so are placed in NEITHER.

Consecutive NLCHARs are treated the same as a single NLCHAR.

8 Concrete syntax

8.1 Introduction

The concrete syntax defines the syntax of the Z language: every sentence of the Z language is recognised by this syntax, and all sentences recognised by this syntax are sentences of the Z language. The concrete syntax is written in terms of the tokens generated by the lexis (clause 7). There are no terminal symbols within this syntax, so as to establish a formal connection with that lexis. Sequences of tokens that are not recognised by this syntax are not sentences of the Z language and are thus excluded from consideration by subsequent phases and so are not given a semantics by this International Standard.

A parser conforming to this concrete syntax converts a concrete sentence to a parse tree.

The non-terminal symbols of the concrete syntax that are written as mathematical symbols or are entirely `CAPITALIZED` or Roman are Z tokens defined in the lexis (claue 7). The other non-terminal symbols are written in `MixedCase` and are defined within the concrete syntax.

8.2 Formal definition of concrete syntax


```
Specification     = { Section }                                          (* sectioned specification *)
                  | { Paragraph }                                        (* anonymous specification *)
                  ;

Section           = ZED , section , NAME , parents , [ NAME , { ,-tok , NAME } ] , END ,
                      { Paragraph }                                      (* inheriting section *)
                  | ZED , section , NAME , END , { Paragraph }           (* base section *)
                  ;

Paragraph         = ZED , [-tok , NAME , { ,-tok , NAME } , ]-tok , END  (* given types *)
                  | AX , SchemaText , END                                (* axiomatic description *)
                  | SCH , NAME , SchemaText , END                        (* schema definition *)
                  | GENAX , [-tok , Formals , ]-tok , SchemaText , END
                                                                         (* generic axiomatic description *)
                  | GENSCH , NAME , [-tok , Formals , ]-tok , SchemaText , END
                                                                         (* generic schema definition *)
                  | ZED , DeclName , == , Expression , END               (* horizontal definition *)
                  | ZED , NAME , [-tok , Formals , ]-tok , == , Expression , END
                                                                         (* generic horizontal definition *)
                  | ZED , GenName , == , Expression , END                (* generic operator definition *)
                  | ZED , Freetype , { & , Freetype } , END              (* free types *)
                  | ZED , ⊢? , Predicate , END                           (* conjecture *)
                  | ZED , [-tok , Formals , ]-tok , ⊢? , Predicate , END (* generic conjecture *)
                  | ZED , OperatorTemplate , END                         (* operator template *)
                  ;

Freetype          = NAME , ::= , Branch , { |-tok , Branch } ;           (* free type *)

Branch            = DeclName , [ ⟨⟨ , Expression , ⟩⟩ ] ;                (* element or injection *)

Formals           = NAME , { ,-tok , NAME } ;                            (* generic parameters *)

Predicate         = Predicate , NL , Predicate                           (* newline conjunction *)
                  | Predicate , ; -tok , Predicate                       (* semicolon conjunction *)
                  | ∀ , SchemaText , • , Predicate                       (* universal quantification *)
                  | ∃ , SchemaText , • , Predicate                       (* existential quantification *)
                  | ∃1 , SchemaText , • , Predicate                      (* unique existential quantification *)
                  | Predicate , ⇔ , Predicate                            (* equivalence *)
                  | Predicate , ⇒ , Predicate                            (* implication *)
                  | Predicate , ∨ , Predicate                            (* disjunction *)
                  | Predicate , ∧ , Predicate                            (* conjunction *)
                  | ¬ , Predicate                                        (* negation *)
```

```
                | Relation                                          (* relation operator application *)
                | Expression                                       (* schema predicate *)
                | true                                             (* truth *)
                | false                                            (* falsity *)
                | (-tok , Predicate , )-tok                        (* parenthesized predicate *)
                ;

Expression      = ∀ , SchemaText , • , Expression                  (* schema universal quantification *)
                | ∃ , SchemaText , • , Expression                  (* schema existential quantification *)
                | ∃1 , SchemaText , • , Expression                 (* schema unique existential quantification *)
                | λ , SchemaText , • , Expression                  (* function construction *)
                | μ , SchemaText , • , Expression                  (* definite description *)
                | let , DeclName , == , Expression ,
                      { ;-tok , DeclName , == , Expression } ,
                          • , Expression                           (* substitution expression *)
                | Expression , ⇔ , Expression                      (* schema equivalence *)
                | Expression , ⇒ , Expression                      (* schema implication *)
                | Expression , ∨ , Expression                      (* schema disjunction *)
                | Expression , ∧ , Expression                      (* schema conjunction *)
                | ¬ , Expression                                   (* schema negation *)
                | if , Predicate , then , Expression , else , Expression   (* conditional *)
                | Expression , ⨾ , Expression                      (* schema composition *)
                | Expression , ≫ , Expression                      (* schema piping *)
                | Expression , \ , (-tok , DeclName , { ,-tok , DeclName } , )-tok
                                                                   (* schema hiding *)
                | Expression , ↾ , Expression                      (* schema projection *)
                | pre , Expression                                 (* schema precondition *)
                | Expression , × , Expression , { × , Expression } (* Cartesian product *)
                | ℙ , Expression                                   (* powerset *)
                | Application                                      (* function and generic operator application *)
                | Expression , Expression                          (* application *)
                | Expression , STROKE                              (* schema decoration *)
                | Expression , [-tok , DeclName , / , DeclName ,
                      { ,-tok , DeclName , / , DeclName } , ]-tok  (* schema renaming *)
                | Expression , . , RefName                         (* binding selection *)
                | Expression , . , NUMERAL                         (* tuple selection *)
                | θ , Expression , { STROKE }                      (* binding construction *)
                | RefName                                          (* reference *)
                | RefName , [-tok , Expression , { ,-tok , Expression } , ]-tok
                                                                   (* generic instantiation *)
                | NUMERAL                                          (* number literal *)
                | {-tok , [ Expression , { ,-tok , Expression } ] , }-tok   (* set extension *)
                | {-tok , SchemaText , • , Expression , }-tok      (* set comprehension *)
                | ( ( {-tok , SchemaText , }-tok )  — ( {-tok , }-tok ) )
                                  — ( {-tok , Expression , }-tok )
                                                                   (* characteristic set comprehension *)
                | ( [-tok , SchemaText , ]-tok )  — ( [-tok , Expression , ]-tok )
                                                                   (* schema construction *)
                | ⦇ , [ DeclName , == , Expression ,
                      { ,-tok , DeclName , == , Expression } ] , ⦈   (* binding extension *)
                | (-tok , Expression , ,-tok , Expression , { ,-tok , Expression } , )-tok
                                                                   (* tuple extension *)
                | (-tok , μ , SchemaText , )-tok                   (* characteristic definite description *)
```



```
                | (-tok , Expression , )-tok                    (* parenthesized expression *)
                ;

SchemaText      = [ DeclPart ] , [ |-tok , Predicate ] ;

DeclPart        = Declaration , { ( ;-tok | NL ) , Declaration } ;

Declaration     = DeclName , { ,-tok , DeclName } , : , Expression
                | DeclName , == , Expression
                | Expression
                ;

OperatorTemplate = relation , Template
                | function , CategoryTemplate
                | generic , CategoryTemplate
                ;

CategoryTemplate = PrefixTemplate
                | PostfixTemplate
                | Prec , Assoc , InfixTemplate
                | NofixTemplate
                ;

Prec            = NUMERAL ;

Assoc           = leftassoc
                | rightassoc
                ;

Template        = PrefixTemplate
                | PostfixTemplate
                | InfixTemplate
                | NofixTemplate
                ;

PrefixTemplate  = (-tok , PrefixName , )-tok
                | (-tok , ℙ , _ , )-tok
                ;

PostfixTemplate = (-tok , PostfixName , )-tok ;

InfixTemplate   = (-tok , InfixName , )-tok ;

NofixTemplate   = (-tok , NofixName , )-tok ;

DeclName        = NAME
                | OpName
                ;

RefName         = NAME
                | (-tok , OpName , )-tok
                ;

OpName          = PrefixName
                | PostfixName
                | InfixName
                | NofixName
                ;
```



```
PrefixName      = PRE , _
                | PREP , _
                | L , { _ , ES | ,, , SS } , ( _ , ERE | ,, , SRE ) , _
                | LP , { _ , ES | ,, , SS } , ( _ , EREP | ,, , SREP ) , _
                ;

PostfixName     = _ , POST
                | _ , POSTP
                | _ , EL , { _ , ES | ,, , SS } , ( _ , ER | ,, , SR )
                | _ , ELP , { _ , ES | ,, , SS } , ( _ , ERP | ,, , SRP )
                ;

InfixName       = _ , I , _
                | _ , IP , _
                | _ , EL , { _ , ES | ,, , SS } , ( _ , ERE | ,, , SRE ) , _
                | _ , ELP , { _ , ES | ,, , SS } , ( _ , EREP | ,, , SREP ) , _
                ;

NofixName       = L , { _ , ES | ,, , SS } , ( _ , ER | ,, , SR )
                | LP , { _ , ES | ,, , SS } , ( _ , ERP | ,, , SRP )
                ;

GenName         = PrefixGenName
                | PostfixGenName
                | InfixGenName
                | NofixGenName
                ;

PrefixGenName   = PRE , NAME
                | L , { NAME , ( ES | SS ) } , NAME , ( ERE | SRE ) , NAME
                ;

PostfixGenName  = NAME , POST
                | NAME , EL , { NAME , ( ES | SS ) } , NAME , ( ER | SR )
                ;

InfixGenName    = NAME , I , NAME
                | NAME , EL , { NAME , ( ES | SS ) } , NAME , ( ERE | SRE ) , NAME
                ;

NofixGenName    = L , { NAME , ( ES | SS ) } , NAME , ( ER | SR ) ;

Relation        = PrefixRel
                | PostfixRel
                | InfixRel
                | NofixRel
                ;

PrefixRel       = PREP , Expression
                | LP , ExpSep , ( Expression , EREP | ExpressionList , SREP ) , Expression
                ;

PostfixRel      = Expression , POSTP
                | Expression , ELP , ExpSep , ( Expression , ERP | ExpressionList , SRP )
                ;
```



```
InfixRel          = Expression , ( ∈ | =-tok | IP ) , Expression ,
                        { ( ∈ | =-tok | IP ) , Expression }
                  | Expression , ELP , ExpSep ,
                      ( Expression , EREP | ExpressionList , SREP ) , Expression
                  ;

NofixRel          = LP , ExpSep , ( Expression , ERP | ExpressionList , SRP ) ;

Application       = PrefixApp
                  | PostfixApp
                  | InfixApp
                  | NofixApp
                  ;

PrefixApp         = PRE , Expression
                  | L , ExpSep , ( Expression , ERE | ExpressionList , SRE ) , Expression
                  ;

PostfixApp        = Expression , POST
                  | Expression , EL , ExpSep , ( Expression , ER | ExpressionList , SR )
                  ;

InfixApp          = Expression , I , Expression
                  | Expression , EL , ExpSep ,
                      ( Expression , ERE | ExpressionList , SRE ) , Expression
                  ;

NofixApp          = L , ExpSep , ( Expression , ER | ExpressionList , SR ) ;

ExpSep            = { Expression , ES | ExpressionList , SS } ;

ExpressionList    = [ Expression , { ,-tok , Expression } ] ;
```

8.3 User-defined operators

A template can reuse the same words as other templates in the same scope, so long as the same token is associated with a word by all templates that use that word. A template's whole operator name shall be different from those of other templates in the same scope. All templates in a scope that use the same word shall have the same precedence.

EXAMPLE 1 These two templates reuse *a* acceptably.

generic 30 leftassoc (_ *a* _ *b* _)
generic 30 leftassoc (_ *a* _ *c* _)

EXAMPLE 2 These two templates conflict unacceptably, because the whole operator names are identical.

function 10 leftassoc (_ *d* _)
generic 20 leftassoc (_ *d* _)

EXAMPLE 3 These two templates conflict unacceptably, because they reuse *a* and have different precedences.

generic 30 leftassoc (_ *a* _ *b* _)
generic 40 leftassoc (_ *a* _ *c* _)

In the `Template` rule's auxiliaries, the name of each of the operator tokens shall not have any `STROKE`s.

NOTE 1 This is so that any common decoration of those words can be treated as an application of a decorated instance of that operator.

All prefix operators are right associative. All postfix operators are left associative. Different associativities shall not be used at the same precedence level by operator template paragraphs in the same scope.

EXAMPLE 4 These two templates conflict unacceptably, because they have the same precedence but different associativities.

function 50 leftassoc $(\_\ e\ \_)$
function 50 rightassoc $(\_\ f\ \_)$

Table 31 defines the relative precedences of the productions of `Expression` and `Predicate`. The rows in the table are ordered so that the entries with higher precedence (and so that bind more strongly) appear nearer the top of the table than those with lower precedence (that bind more weakly). Associativity has significance only in determining the nesting of applications involving non-associative operators of the same precedence. Explicitly-defined infix function and generic operator applications have a range of precedences specified numerically in the corresponding operator template paragraph. Cartesian product expressions have precedence value 8 within that numeric range.

NOTE 2 The order of productions in the `Predicate` and `Expression` rules is based roughly on the precedences of their operators. Some productions have the same precedence as their neighbours, and so the separate table of operator precedences is necessary.

NOTE 3 One way of parsing nested operator applications at different user-defined levels of precedence and associativity is explained by Lalonde and des Rivieres [9]. Lalonde and des Rivieres approach is to "use a grammar ... that describes expressions without regard to the precedence [and associativity] of the operators", and then to transform the resulting parse tree to take account of precedence and associativity. Assuming the grammar uses left associativity for all operators, the transformations to take into account precedence and associativity where necessary are as follows.

$$\begin{aligned}(e_1\ \mathit{infix}_1\ e_2)\ \mathit{infix}_2\ e_3 &\Longrightarrow e_1\ \mathit{infix}_1\ (e_2\ \mathit{infix}_2\ e_3)\\ (e_1\ \mathit{infix}\ e_2)\ \mathit{post} &\Longrightarrow e_1\ \mathit{infix}\ (e_2\ \mathit{post})\\ (\mathit{pre}\ e)\ \mathit{post} &\Longrightarrow \mathit{pre}\ (e\ \mathit{post})\end{aligned}$$

If right associativity was instead assumed, then the following transformations would be needed.

$$\begin{aligned}e_1\ \mathit{infix}_1\ (e_2\ \mathit{infix}_2\ e_3) &\Longrightarrow (e_1\ \mathit{infix}_1\ e_2)\ \mathit{infix}_2\ e_3\\ \mathit{pre}\ (e_1\ \mathit{infix}\ e_2) &\Longrightarrow (\mathit{pre}\ e_1)\ \mathit{infix}\ e_2\end{aligned}$$

Using distinct variants of the operator tokens `PRE`|...|`SS` for relational operators from those for function and generic operators allows these transformations to avoid dealing with those notations whose precedences lie between the relations and the functions, such as the schema operations.

NOTE 4 In the `PrefixTemplate` rule, the production for powerset enables explicit definition of $\mathbb{P}_1$ in the mathematical toolkit to coexist with the treatment of $\mathbb{P}$ as a keyword by the lexis.

8.4 Additional syntactic restrictions and notes

`STROKE`s are used in three contexts: within `NAME`s, in binding construction expressions, and in schema decoration expressions. The condition for a `STROKE` to be considered as part of a `NAME` was given in 7.3. Other `STROKE`s are considered to be parts of binding construction expressions if they can be when interpreted from left to right. The schema decoration expression interpretation is considered last.

EXAMPLE 1 In $\theta S'\ '\ '$, the first $'$ is part of the `NAME` S', the second $'$ can be part of the binding construction expression, and the third $'$ is syntactically part of a schema decoration expression, though that will be rejected by the type inference rules as only schemas can be decorated, not bindings.

A predicate can be just an expression, yet the same logical operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists$, $\exists_1$) can be used in both expressions and predicates. Where a predicate is expected, and one of these logical operators is used on expressions, there is an ambiguity: either the whole logical operation is an expression and that expression is used

Table 31 — Operator precedences and associativities

Productions	Associativity
binding construction	
binding selection, tuple selection	
schema renaming	
schema decoration	
application	left
postfix function and generic operator application	
powerset, prefix function and generic operator application	
Cartesian product, infix function and generic operator application	
schema precondition	
schema projection	left
schema hiding	
schema piping	
schema composition	
conditional	
substitution expression	
definite description	
function construction	
relation operator application	
negation	
conjunction	
disjunction	
implication	right
equivalence	
universal, existential and unique existential quantifications	
newline conjunction, semicolon conjunction	

as a predicate, or the whole logical operation is a predicate involving expressions each used as a predicate. This ambiguity is benign, as both interpretations have the same precedence, associativity and meaning.

A section header shall be parsed in its entirety before bringing the declarations and operator templates of its parent sections into scope.

NOTE 1 This prevents surprises when the name of a parent section is the same as the name of an operator defined in another parent.

Application expressions' lack of any keyword token between their juxtaposed expressions is a problem for some implementations of parsers, which may require the parentheses in phrases such as $f\ (seq\ X)$.

NOTE 2 The juxtaposition of two expressions $e_1 e_2$ is always parsed as the application of function e_1 to argument e_2, never as the application of relation e_1 to argument e_2 which in some previous dialects of Z, e.g. King *et al* [8], was equivalent to the relation $e_2 \in e_1$. In Standard Z, membership is the normal form (canonical representation) of all relational predicates (see 12.2.10), and juxtaposition is the normal form of all application expressions (see 12.2.11).

NOTE 3 The syntax of conjectures is deliberately simple. This is so as to be compatible with the syntaxes of sequents as found in as many different theorem provers as possible, while establishing a common form to enable interchange.

NOTE 4 Implementations of parsers commonly inspect the next token from the input stream and have to decide there and then whether that token is another token in an incomplete phrase or whether the current phrase is complete and the token is starting a new phrase. The tokens defined by the lexis (clause 7) are insufficient for such an implementation of a parser.

EXAMPLE 2 At the first comma in the set extension $\{x, y, z\}$ the x shall be seen to be an expression, yet the phrase might yet turn out to be the set comprehension $\{x, y, z : e\}$.

EXAMPLE 3 At the opening square bracket in the application to a schema construction $i\ [e_1;\ e_2]$ the name i shall be seen to be an expression, yet the phrase might yet turn out to be the generic instantiation $i[e_1]$.

One solution to such problems is to try all possible parses and accept the one that works. Another solution is to have the lexer lookahead far enough to be able to distinguish which of the alternative cases is present, and to provide the parser with one of several distinct tokens. Expressions $\{x, y, z\}$ and $\{x, y, z : e\}$ can be distinguished by looking ahead from a comma, over following alternating names (including operator names) and commas, for a : or == token, and using distinct comma tokens depending on whether that is seen or not. Expressions $i\ [e_1;\ e_2]$ and $i[e_1]$ can be distinguished by looking ahead from the open square bracket for the matching closing square bracket, stopping if a ; -tok, :, == or |-tok token is encountered, and using distinct open square bracket tokens for the matched and stopped cases.

NOTE 5 An `ExpressionList` phrase will be regarded as an expression whose value is a sequence (see 12.2.12). In defining operators that take such `ExpressionList`s as arguments, it is convenient to have the operations of *sequence_toolkit* (see B.8) in scope.

NOTE 6 The `ExpressionList` rule is used only in operator applications, not in set extension, tuple extension and generic instantiation expressions, so that the syntactic transformation in 12.2.12 is applied only to operator applications.

8.5 Box renderings

There are two different sets of box renderings in widespread use, as illustrated here. Any particular presentation of a section shall use one set or the other throughout. The middle line shall be omitted when the paragraph has no predicates, but otherwise shall be retained if the paragraph has no declarations. The outlines need be only as wide as the text, but are here shown as wide as the page.

8.5.1 First box rendering

The following four paragraphs illustrate the first of two alternative renderings of box tokens.

An axiomatic paragraph, involving the `AX`, |-tok and `END` tokens, shall have this box rendering.

```
| DeclPart
|-----------
| Predicate
```

A schema paragraph, involving the `SCH`, |-tok and `END` tokens, shall have this box rendering.

```
NAME
| DeclPart
|-----------
| Predicate
```

A generic axiomatic paragraph, involving the `GENAX`, |-tok and `END` tokens, shall have this box rendering.

```
[Formals]
| DeclPart
|-----------
| Predicate
```

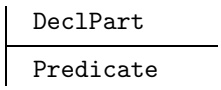
A generic schema paragraph, involving the `GENSCH`, |-tok and `END` tokens, shall have this box rendering.

```
NAME [Formals]
| DeclPart
|-----------
| Predicate
```

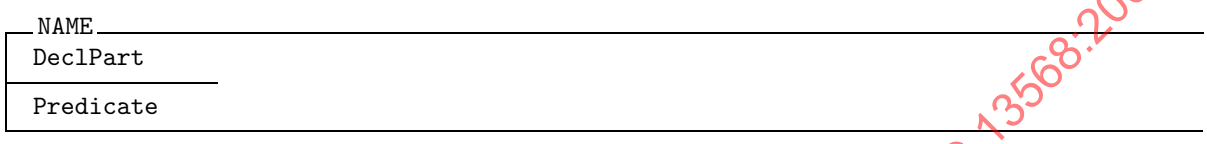

8.5.2 Second box rendering

The following four paragraphs illustrate the second of two alternative renderings of box tokens.

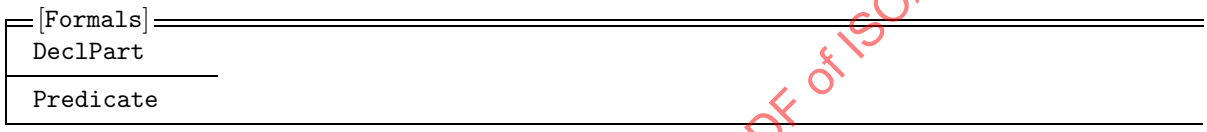
An axiomatic paragraph, involving the AX, |-tok and END tokens, shall have this box rendering.



A schema paragraph, involving the SCH, |-tok and END tokens, shall have this box rendering.



A generic axiomatic paragraph, involving the GENAX, |-tok and END tokens, shall have this box rendering.



A generic schema paragraph, involving the GENSCH, |-tok and END tokens, shall have this box rendering.

9 Characterisation rules

9.1 Introduction

The characterisation rules together map the parse tree of a concrete syntax sentence to the parse tree of an equivalent concrete syntax sentence in which all implicit characteristic tuples have been made explicit.

Only concrete trees that are mapped to different trees are given explicit characterisation rules. The characterisation rules are listed in the same order as the corresponding productions of the concrete syntax.

Characteristic tuples are calculated from schema texts by the metalanguage function *chartuple* (see 9.2).

9.2 Characteristic tuple

A characteristic tuple is computed in two phases: *charac*, which returns a sequence of expressions, and *mktuple*, which converts that sequence into the characteristic tuple.

$$chartuple\ t \quad = \quad mktuple\ (charac\ t)$$

Sequences of expressions are enclosed between metalanguage brackets $\langle$ and $\rangle$, in general $\langle e_1, ..., e_n \rangle$. Two sequences of expressions are concatenated by the $\frown$ operator.

$$\langle e_1, ..., e_n \rangle \frown \langle e_{n+1}, ..., e_{n+m} \rangle \quad = \quad \langle e_1, ..., e_n, e_{n+1}, ..., e_{n+m} \rangle$$

$$
\begin{aligned}
charac\ (d_1;\ ...;\ d_n \mid p) &= charac\ (d_1;\ ...;\ d_n) \\
charac\ (d_1;\ ...;\ d_n) &= charac\ d_1 \frown ... \frown charac\ d_n \qquad where\ n \geq 1 \\
charac\ () &= \langle\!|\ |\!\rangle \\
charac\ (i_1, ..., i_n : e) &= \langle i_1, ..., i_n \rangle \\
charac\ (i == e) &= \langle i \rangle \\
charac\ (e\ ^*) &= \langle \theta\, e^* \rangle
\end{aligned}
$$

$$
\begin{aligned}
mktuple\ \langle e \rangle &= e \\
mktuple\ \langle e_1, ..., e_n \rangle &= (e_1, ..., e_n) \qquad where\ n \geq 2
\end{aligned}
$$

NOTE 1 In the last case of *charac*, the type inference rule in 13.2.6.9 ensures that e is a schema.

NOTE 2 In *mktuple*, the result is a Z expression, so the brackets in its second equation are those of a tuple extension.

NOTE 3 The characteristic tuple operation determines a new phrase for use in a larger phrase. It does not manipulate semantic values or type values. It is more akin to a transformation rule in that it operates on phrases, but unlike a transformation rule it does not replace a phrase by an equivalent phrase. Its definition needs auxiliaries for representing sequences of phrases and concatenating sequences of phrases. The sequence brackets defined in 4.2.11 are not suitable, as they manipulate metalanguage sequences. Rather than invent yet more notation, the usual Z-like notations for sequences are redefined just for this definition.

9.3 Formal definition of characterisation rules

9.3.1 Function construction expression

The value of the function construction expression $\lambda\ t \bullet e$ is the function associating values of the characteristic tuple of t with corresponding values of e.

$$\lambda\ t \bullet e \quad \Longrightarrow \quad \{t \bullet (chartuple\ t, e)\}$$

It is semantically equivalent to the set of pairs representation of that function.

9.3.2 Characteristic set comprehension expression

The value of the characteristic set comprehension expression $\{t\}$ is the set of the values of the characteristic tuple of t.

$$\{t\} \quad \Longrightarrow \quad \{t \bullet chartuple\ t\}$$

It is semantically equivalent to the corresponding set comprehension expression in which the characteristic tuple is made explicit.

9.3.3 Characteristic definite description expression

The value of the characteristic definite description expression $(\mu\ t)$ is the sole value of the characteristic tuple of schema text t.

$$(\mu\ t) \quad \Longrightarrow \quad \mu\ t \bullet chartuple\ t$$

It is semantically equivalent to the corresponding definite description expression in which the characteristic tuple is made explicit.

10 Annotated syntax

10.1 Introduction

The annotated syntax defines a language that includes all sentences that could be produced by application of the syntactic transformation rules (clause 12) to sentences of the concrete syntax (clause 8). This language's set of sentences would be a subset of that defined by the concrete syntax but for introduction of type annotations and use of expressions in place of schema texts.

Like the concrete syntax, this annotated syntax is written in terms of the tokens generated by the lexis; there are no terminal symbols within this syntax. The added type annotation notation uses three tokens not defined in the lexis: `GIVEN`, `GENTYPE`, and ⦂.

Some additional characters that are presumed to be distinct from the characters used in concrete phrases are introduced as follows. The constituent words of operators are glued together to form single names using the character ⋈, which belongs to the `WORDGLUE` character class. (These single names cannot clash with any existing names.) For comparing the bindings of two schemas, one schema is decorated using the character ⋈, which belongs to the `STROKECHAR` character class. (Inclusion of a schema so decorated cannot capture any existing references.) For defining the semantics of types, the names of given sets and generic parameters are decorated with characters ♡ and ♠ respectively, both of which belong to the `STROKECHAR` character class. (This avoids local declarations of the same names making holes in the scopes of the types.)

There are no parentheses in the annotated syntax as defined here. A sentence or phrase of the annotated syntax should be thought of as a tree structure of nested formulae. When presented as linear text, however, the precedences of the concrete syntax may be assumed and parentheses may be inserted to override those precedences. The precedence of the type annotation ⦂ operator is then weaker than all other operators, and the precedences and associativities of the type notations are analogous to those of the concrete notations of similar appearance.

NOTE 1 This annotated syntax permits some verification of the syntactic transformation rules to be performed.

NOTE 2 The annotated syntax is similar to an annotated tree (abstract syntax) used in a tool, but the level of abstraction effected by the characterization rules and syntactic transformation rules might not be appropriate for a tool.

10.2 Formal definition of annotated syntax

```
Specification     = { Section } ;                                   (* sectioned specification *)

Section           = ZED , section , NAME , parents , [ NAME , { ,-tok , NAME } ] , END ,
                        { Paragraph } , [ ⦂ , SectTypeEnv ] ;          (* inheriting section *)

Paragraph         = ZED , [-tok , NAME , { ,-tok , NAME } , ]-tok , END ,
                        [ ⦂ , Signature ]                                  (* given types *)
                  | AX , Expression , END ,
                        [ ⦂ , Signature ]                          (* axiomatic description *)
                  | GENAX , [-tok , NAME , { ,-tok , NAME } , ]-tok , Expression , END ,
                        [ ⦂ , Signature ]                 (* generic axiomatic description *)
                  | ZED , NAME , ::= , NAME , [ ⟪ , Expression , ⟫ ] ,
                      { |-tok , NAME , [ ⟪ , Expression , ⟫ ] } ,
                      { & , NAME , ::= , NAME , [ ⟪ , Expression , ⟫ ] ,
                      { |-tok , NAME , [ ⟪ , Expression , ⟫ ] } } , END ,
                        [ ⦂ , Signature ]                                (* free types *)
                  | ZED , ⊢? , Predicate , END ,
                        [ ⦂ , Signature ]                               (* conjecture *)
                  | ZED , [-tok , NAME , { ,-tok , NAME } , ]-tok , ⊢? , Predicate , END ,
                        [ ⦂ , Signature ]                       (* generic conjecture *)
                  ;
```



```
Predicate       = Expression , ∈ , Expression                          (* membership *)
                | true                                                 (* truth *)
                | ¬ , Predicate                                        (* negation *)
                | Predicate , ∧ , Predicate                            (* conjunction *)
                | ∀ , Expression , • , Predicate                       (* universal quantification *)
                | ∃1 , Expression , • , Predicate                      (* unique existential quantification *)
                ;

Expression      = NAME ,
                      [ ⦂ , Type ]                                     (* reference *)
                | NAME , [-tok , Expression , { ,-tok , Expression } , ]-tok ,
                      [ ⦂ , Type ]                                     (* generic instantiation *)
                | {-tok , [ Expression , { ,-tok , Expression } ] , }-tok ,
                      [ ⦂ , Type ]                                     (* set extension *)
                | {-tok , Expression , • , Expression , }-tok ,
                      [ ⦂ , Type ]                                     (* set comprehension *)
                | ℙ , Expression ,
                      [ ⦂ , Type ]                                     (* powerset *)
                | (-tok , Expression , ,-tok , Expression , { ,-tok , Expression } , )-tok ,
                      [ ⦂ , Type ]                                     (* tuple extension *)
                | Expression , . , NUMERAL ,
                      [ ⦂ , Type ]                                     (* tuple selection *)
                | ⦇ , NAME , == , Expression ,
                      { ,-tok , NAME , == , Expression } , ⦈ ,
                          [ ⦂ , Type ]                                 (* binding extension *)
                | θ , Expression , { STROKE } ,
                      [ ⦂ , Type ]                                     (* binding construction *)
                | Expression , . , NAME ,
                      [ ⦂ , Type ]                                     (* binding selection *)
                | Expression , Expression ,
                      [ ⦂ , Type ]                                     (* application *)
                | μ , Expression , • , Expression ,
                      [ ⦂ , Type ]                                     (* definite description *)
                | [-tok , NAME , : , Expression , ]-tok ,
                      [ ⦂ , Type ]                                     (* variable construction *)
                | [-tok , Expression , |-tok , Predicate , ]-tok ,
                      [ ⦂ , Type ]                                     (* schema construction *)
                | ¬ , Expression ,
                      [ ⦂ , Type ]                                     (* schema negation *)
                | Expression , ∧ , Expression ,
                      [ ⦂ , Type ]                                     (* schema conjunction *)
                | Expression , \ , (-tok , NAME , { ,-tok , NAME } , )-tok ,
                      [ ⦂ , Type ]                                     (* schema hiding *)
                | ∀ , Expression , • , Expression ,
                      [ ⦂ , Type ]                                     (* schema universal quantification *)
                | ∃1 , Expression , • , Expression ,
                      [ ⦂ , Type ]                                     (* schema unique existential quantification *)
                | Expression , [-tok , NAME , / , NAME ,
                      { ,-tok , NAME , / , NAME } , ]-tok ,
                          [ ⦂ , Type ]                                 (* schema renaming *)
                | pre , Expression ,
                      [ ⦂ , Type ]                                     (* schema precondition *)
```



```
                | Expression , ⨟ , Expression ,
                      [ ⨟ , Type ]                          (* schema composition *)
                | Expression , ≫ , Expression ,
                      [ ⨟ , Type ]                          (* schema piping *)
                | Expression , STROKE ,
                      [ ⨟ , Type ]                          (* schema decoration *)
                ;

SectTypeEnv     = [ NAME , : , (-tok , NAME , ,-tok , Type , )-tok ,
                    { ;-tok , NAME , : , (-tok , NAME , ,-tok , Type , )-tok } ] ;

Type            = [-tok , NAME , { ,-tok , NAME } , ]-tok ,
                      Type2 , [ ,-tok , Type2 ]             (* generic type *)
                | Type2
                ;
Type2           = GIVEN , NAME                              (* given type *)
                | GENTYPE , NAME                            (* generic parameter type *)
                | ℙ , Type2                                 (* powerset type *)
                | Type2 , × , Type2 , { × , Type2 }         (* Cartesian product type *)
                | [-tok , Signature , ]-tok                 (* schema type *)
                ;

Signature       = [ NAME , : , Type , { ;-tok , NAME , : , Type } ]
                | ϵ                                         (* empty signature *)
                ;
```

10.3 Notes

NOTE 1 More free types than necessary are permitted by this syntax: as a result of the syntactic transformation in 12.2.3.5, all elements appear before all injections.

NOTE 2 The only signatures that contain generic types are those in the annotations of generic axiomatic description paragraphs.

11 Prelude

11.1 Introduction

The prelude is a Z section. It is an implicit parent of every other section. It assists in defining the meaning of number literal expressions (see 12.2.6.9), and the list arguments of operators (see 12.2.12), via syntactic transformation rules later in this International Standard. The prelude is presented here using the mathematical lexis.

11.2 Formal definition of prelude

section *prelude*

The section is called *prelude* and has no parents.

NOTE 1

generic $(\mathbb{P}\_)$

$\mathbb{P}$ has already been introduced in this International Standard (see 8.3), so this operator template is not necessary. However, it may be a convenient way of introducing to a tool the association of $\mathbb{P}$ with the appropriate token and precedence, especially in preparation for the toolkit's $\mathbb{P}_1$ (see B.3.6). A tool may introduce $\times$ here similarly, that being the only other Z core notation whose precedence lies amongst those of user-defined operators.

$[\mathbb{A}]$

The given type $\mathbb{A}$, pronounced "arithmos", provides a supply of values for use in specifying number systems.

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

The set of natural numbers, $\mathbb{N}$, is a subset of $\mathbb{A}$.

$$number\_literal\_0 : \mathbb{N}$$
$$number\_literal\_1 : \mathbb{N}$$

0 and 1 are natural numbers, all uses of which are transformed to references to these declarations (see 12.2.6.9).

function 30 leftassoc $(\_ + \_)$

$$\_ + \_ : \mathbb{P}\,((\mathbb{A} \times \mathbb{A}) \times \mathbb{A})$$

$$\forall\, m, n : \mathbb{N} \bullet \exists_1\, p : (\_ + \_) \bullet p.1 = (m, n)$$
$$\forall\, m, n : \mathbb{N} \bullet m + n \in \mathbb{N}$$
$$\forall\, m, n : \mathbb{N} \mid m + 1 = n + 1 \bullet m = n$$
$$\forall\, m : \mathbb{N} \bullet \neg\, m + 1 = 0$$
$$\forall\, w : \mathbb{P}\ \mathbb{N} \mid 0 \in w \land (\forall\, y : w \bullet y + 1 \in w) \bullet w = \mathbb{N}$$
$$\forall\, m : \mathbb{N} \bullet m + 0 = m$$
$$\forall\, m, n : \mathbb{N} \bullet m + (n + 1) = m + n + 1$$

Addition is defined here for natural numbers. (It is extended to integers in the mathematical toolkit, annex B.) Addition is a function. The sum of two natural numbers is a natural number. The operation of adding 1 is an injection on natural numbers, and produces a result different from 0. There is an induction constraint that all natural numbers are either 0 or are formed by adding 1 to another natural number. 0 is an identity of addition. Addition is associative.

NOTE 2 The definition of addition is equivalent to the following definition, which is written using notation from the mathematical toolkit (and so is unsuitable as the normative definition here).

$$\_ + \_ : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A}$$

$$(\mathbb{N} \times \mathbb{N}) \lhd (\_ + \_) \in (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$$
$$\lambda\, n : \mathbb{N} \bullet n + 1 \in \mathbb{N} \rightarrowtail \mathbb{N}$$
$$disjoint\langle \{0\}, \{n : \mathbb{N} \bullet n + 1\} \rangle$$
$$\forall\, w : \mathbb{P}\,\mathbb{N} \mid \{0\} \cup \{n : w \bullet n + 1\} \subseteq w \bullet w = \mathbb{N}$$
$$\forall\, m : \mathbb{N} \bullet m + 0 = m$$
$$\forall\, m, n : \mathbb{N} \bullet m + (n + 1) = m + n + 1$$

12 Syntactic transformation rules

12.1 Introduction

The syntactic transformation rules together map the parse tree of a concrete syntax sentence to the parse tree of a semantically equivalent annotated syntax sentence. The resulting annotated parse trees may refer to definitions of the prelude.

Although exhaustive application of the syntactic transformation rules produces annotated parse trees, individual syntactic transformation rules can produce a mixture of concrete and annotated notation. Explicit distinction of the two is not done, as it would be cumbersome and detract from readability.

Only concrete trees that are not in the annotated syntax are given explicit syntactic transformation rules. The syntactic transformation rules are listed in the same order as the corresponding productions of the concrete syntax. Where an individual concrete syntax production is expressed using alternations, a separate syntactic transformation rule is given for each alternative.

All applications of syntactic transformation rules that generate new declarations shall choose the names of those declarations to be such that they do not capture references during subsequent typechecking.

Rules that generate type annotations generate annotations with fresh variables each time they are applied.

12.2 Formal definition of syntactic transformation rules

12.2.1 Specification

12.2.1.1 Anonymous specification

The anonymous specification D_1 ... D_n is semantically equivalent to the sectioned specification comprising a single section containing those paragraphs with the mathematical toolkit of annex B as its parent.

$$D_1 \ldots D_n \implies \textit{Mathematical toolkit}\ \texttt{ZED}\ \text{section}\ \textit{Specification}\ \text{parents}\ \textit{standard\_toolkit}\ \texttt{END}\ D_1 \ldots D_n$$

In this transformation, *Mathematical toolkit* denotes the entire text of annex B. The name given to the section is not important: it need not be *Specification*, though it shall not be *prelude* or that of any section of the mathematical toolkit.

12.2.2 Section

12.2.2.1 Base section

The base section ZED section i END D_1 ... D_n is semantically equivalent to the inheriting section that inherits from no parents (bar *prelude*).

$$\texttt{ZED}\ \text{section}\ i\ \texttt{END}\ D_1 \ldots D_n \implies \texttt{ZED}\ \text{section}\ i\ \text{parents}\ \texttt{END}\ D_1 \ldots D_n$$

12.2.3 Paragraph

12.2.3.1 Schema definition paragraph

The schema definition paragraph SCH i t END introduces the global name i, associating it with the schema that is the value of t.

$$\texttt{SCH}\ i\ t\ \texttt{END} \implies \texttt{AX}\ [i == t]\ \texttt{END}$$

The paragraph is semantically equivalent to the axiomatic description paragraph whose sole declaration associates the schema's name with the expression resulting from syntactic transformation of the schema text.

12.2.3.2 Generic schema definition paragraph

The generic schema definition paragraph GENSCH i $[i_1, ..., i_n]$ t END can be instantiated to produce a schema definition paragraph.

$$\texttt{GENSCH}\ i\ [i_1, ..., i_n]\ t\ \texttt{END} \implies \texttt{GENAX}\ [i_1, ..., i_n]\ [i == t]\ \texttt{END}$$

It is semantically equivalent to the generic axiomatic description paragraph with the same generic parameters and whose sole declaration associates the schema's name with the expression resulting from syntactic transformation of the schema text.

12.2.3.3 Horizontal definition paragraph

The horizontal definition paragraph $\texttt{ZED}\ i == e\ \texttt{END}$ introduces the global name i, associating with it the value of e.

$$\texttt{ZED}\ i == e\ \texttt{END} \implies \texttt{AX}\ [i == e]\ \texttt{END}$$

It is semantically equivalent to the axiomatic description paragraph that introduces the same single declaration.

12.2.3.4 Generic horizontal definition paragraph

The generic horizontal definition paragraph $\texttt{ZED}\ i\ [i_1, ..., i_n] == e\ \texttt{END}$ can be instantiated to produce a horizontal definition paragraph.

$$\texttt{ZED}\ i\ [i_1, ..., i_n] == e\ \texttt{END} \implies \texttt{GENAX}\ [i_1, ..., i_n]\ [i == e]\ \texttt{END}$$

It is semantically equivalent to the generic axiomatic description paragraph with the same generic parameters and that introduces the same single declaration.

12.2.3.5 Free types paragraph

The transformation of free types paragraphs is done in two stages. First, the branches are permuted to bring elements to the front and injections to the rear.

$$... \mid g\langle\langle e \rangle\rangle \mid h \mid ... \implies ... \mid h \mid g\langle\langle e \rangle\rangle \mid ...$$

Exhaustive application of this syntactic transformation rule effects a sort.

The second stage requires implicit generic instantiations to have been filled in, which is done during typechecking (see 13.2.3.3). Hence that second stage is delayed until after typechecking, where it appears in the form of a semantic transformation rule (see 14.2.3.1).

12.2.4 Operator template

There are no syntactic transformation rules for operator template paragraphs; rather, operator template paragraphs determine which syntactic transformation rule to use for each phrase that refers to or applies an operator.

12.2.5 Predicate

12.2.5.1 Newline conjunction predicate

The newline conjunction predicate $p_1\ \texttt{NL}\ p_2$ is *true* if and only if both its predicates are *true*.

$$p_1\ \texttt{NL}\ p_2 \implies p_1 \land p_2$$

It is semantically equivalent to the conjunction predicate $p_1 \land p_2$.

12.2.5.2 Semicolon conjunction predicate

The semicolon conjunction predicate $p_1;\ p_2$ is *true* if and only if both its predicates are *true*.

$$p_1;\ p_2 \implies p_1 \land p_2$$

It is semantically equivalent to the conjunction predicate $p_1 \land p_2$.

12.2.5.3 Existential quantification predicate

The existential quantification predicate $\exists\ t \bullet p$ is *true* if and only if p is *true* for at least one value of t.

$$\exists\ t \bullet p \implies \neg\ \forall\ t \bullet \neg\ p$$

It is semantically equivalent to p being *false* for not all values of t.

12.2.5.4 Equivalence predicate

The equivalence predicate $p_1 \Leftrightarrow p_2$ is *true* if and only if both p_1 and p_2 are *true* or neither is *true*.

$$p_1 \Leftrightarrow p_2 \quad \Longrightarrow \quad (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$$

It is semantically equivalent to each of p_1 and p_2 being *true* if the other is *true*.

12.2.5.5 Implication predicate

The implication predicate $p_1 \Rightarrow p_2$ is *true* if and only if p_2 is *true* if p_1 is *true*.

$$p_1 \Rightarrow p_2 \quad \Longrightarrow \quad \neg\, p_1 \vee p_2$$

It is semantically equivalent to p_1 being *false* disjoined with p_2 being *true*.

12.2.5.6 Disjunction predicate

The disjunction predicate $p_1 \vee p_2$ is *true* if and only if at least one of p_1 and p_2 is *true*.

$$p_1 \vee p_2 \quad \Longrightarrow \quad \neg\, (\neg\, p_1 \wedge \neg\, p_2)$$

It is semantically equivalent to not both of p_1 and p_2 being *false*.

12.2.5.7 Schema predicate

The schema predicate e is *true* if and only if the binding of the names in the signature of schema e satisfies the constraints of that schema.

$$e \quad \Longrightarrow \quad \theta\, e \in e$$

It is semantically equivalent to the binding constructed by θe being a member of the set denoted by schema e.

12.2.5.8 Falsity predicate

The falsity predicate false is never *true*.

$$\text{false} \quad \Longrightarrow \quad \neg\ \text{true}$$

It is semantically equivalent to the negation of true.

12.2.5.9 Parenthesized predicate

The parenthesized predicate (p) is *true* if and only if p is *true*.

$$(p) \quad \Longrightarrow \quad p$$

It is semantically equivalent to p.

12.2.6 Expression

12.2.6.1 Schema existential quantification expression

The value of the schema existential quantification expression $\exists\ t \bullet e$ is the set of bindings of schema e restricted to exclude names that are in the signature of t, for at least one binding of the schema t.

$$\exists\ t \bullet e \quad \Longrightarrow \quad \neg\ \forall\ t \bullet \neg\ e$$

It is semantically equivalent to the result of applying de Morgan's law.

12.2.6.2 Substitution expression

The value of the substitution expression $\text{let}\, i_1 == e_1;\ \ldots;\ i_n == e_n \bullet e$ is the value of e when all of its references to the names have been substituted by the values of the corresponding expressions.

$$\text{let}\, i_1 == e_1;\ \ldots;\ i_n == e_n \bullet e \quad \Longrightarrow \quad \mu\ i_1 == e_1;\ \ldots;\ i_n == i_n \bullet e$$

It is semantically equivalent to the similar definite description expression.

12.2.6.3 Schema equivalence expression

The value of the schema equivalence expression $e_1 \Leftrightarrow e_2$ is that schema whose signature is the union of those of schemas e_1 and e_2, and whose bindings are those whose relevant restrictions are either both or neither in e_1 and e_2.

$$e_1 \Leftrightarrow e_2 \quad \Longrightarrow \quad (e_1 \Rightarrow e_2) \wedge (e_2 \Rightarrow e_1)$$

It is semantically equivalent to the schema conjunction of the schema implication $e_1 \Rightarrow e_2$ with the schema implication $e_2 \Rightarrow e_1$.

12.2.6.4 Schema implication expression

The value of the schema implication expression $e_1 \Rightarrow e_2$ is that schema whose signature is the union of those of schemas e_1 and e_2, and whose bindings are those whose restriction to the signature of e_2 is in the value of e_2 if its restriction to the signature of e_1 is in the value of e_1.

$$e_1 \Rightarrow e_2 \quad \Longrightarrow \quad \neg\, e_1 \vee e_2$$

It is semantically equivalent to the schema disjunction of the schema negation $\neg\, e_1$ with e_2.

12.2.6.5 Schema disjunction expression

The value of the schema disjunction expression $e_1 \vee e_2$ is that schema whose signature is the union of those of schemas e_1 and e_2, and whose bindings are those whose restriction to the signature of e_1 is in the value of e_1 or its restriction to the signature of e_2 is in the value of e_2.

$$e_1 \vee e_2 \quad \Longrightarrow \quad \neg\, (\neg\, e_1 \wedge \neg\, e_2)$$

It is semantically equivalent to the schema negation of the schema conjunction of the schema negations of e_1 and e_2.

12.2.6.6 Conditional expression

The value of the conditional expression if p then e_1 else e_2 is the value of e_1 if p is *true*, and is the value of e_2 if p is *false*.

$$\text{if } p \text{ then } e_1 \text{ else } e_2 \quad \Longrightarrow \quad \mu\ i : \{e_1, e_2\} \mid p \wedge i = e_1 \vee \neg\, p \wedge i = e_2 \bullet i$$

It is semantically equivalent to the definite description expression whose value is either that of e_1 or that of e_2 such that if p is *true* then it is e_1 or if p is *false* then it is e_2.

12.2.6.7 Schema projection expression

The value of the schema projection expression $e_1 \restriction e_2$ is the schema that is like the conjunction $e_1 \wedge e_2$ but whose signature is restricted to just that of schema e_2.

$$e_1 \restriction e_2 \quad \Longrightarrow \quad \{e_1;\ e_2 \bullet \theta\, e_2\}$$

It is semantically equivalent to that set of bindings of names in the signature of e_2 to values that satisfy the constraints of both e_1 and e_2.

12.2.6.8 Cartesian product expression

The value of the Cartesian product expression $e_1 \times ... \times e_n$ is the set of all tuples whose components are members of the corresponding sets that are the values of its expressions.

$$e_1 \times ... \times e_n \quad \Longrightarrow \quad \{i_1 : e_1;\ ...;\ i_n : e_n \bullet (i_1, ..., i_n)\}$$

It is semantically equivalent to the set comprehension expression that declares members of the sets and assembles those members into tuples.

12.2.6.9 Number literal expression

The value of the multiple-digit number literal expression bc is the number that it denotes.

$$bc \implies b+b+b+b+b+b+ \\ b+b+b+b+b+c$$

It is semantically equivalent to the sum of ten repetitions of the number literal expression b formed from all but the last digit, added to that last digit.

$$\begin{aligned} 0 &\implies number\_literal\_0 \\ 1 &\implies number\_literal\_1 \\ 2 &\implies 1+1 \\ 3 &\implies 2+1 \\ 4 &\implies 3+1 \\ 5 &\implies 4+1 \\ 6 &\implies 5+1 \\ 7 &\implies 6+1 \\ 8 &\implies 7+1 \\ 9 &\implies 8+1 \end{aligned}$$

The number literal expressions 0 and 1 are semantically equivalent to *number_literal_0* and *number_literal_1* respectively as defined in section *prelude*. The remaining digits are defined as being successors of their predecessors, using the function + as defined in section *prelude*.

NOTE These syntactic transformations are applied only to `NUMERAL` tokens that form number literal expressions, not to other `NUMERAL` tokens (those in tuple selection expressions and operator template paragraphs), as those other occurrences of `NUMERAL` do not have semantic values associated with them.

12.2.6.10 Schema construction expression

The value of the schema construction expression $[t]$ is that schema whose signature is the names declared by the schema text t, and whose bindings are those that satisfy the constraints in t.

$$[t] \implies t$$

It is semantically equivalent to the schema resulting from syntactic transformation of the schema text t.

12.2.6.11 Parenthesized expression

The value of the parenthesized expression (e) is the value of expression e.

$$(e) \implies e$$

It is semantically equivalent to e.

12.2.7 Schema text

There is no separate schema text class in the annotated syntax: all concrete schema texts are transformed to expressions.

12.2.7.1 `Declaration`

Each declaration is transformed to an equivalent expression.

A constant declaration is equivalent to a variable declaration in which the variable ranges over a singleton set.

$$i == e \implies i : \{e\}$$

A comma-separated multiple declaration is equivalent to the schema conjunction of variable construction expressions in which all variables are constrained to be of the same type.

$$i_1, ..., i_n : e \implies [i_1 : e \mid \alpha_1] \land ... \land [i_n : e \mid \alpha_1]$$

12.2.7.2 DeclPart

Each declaration part is transformed to an equivalent expression.

$$d_1; ...; d_n \implies d_1 \land ... \land d_n$$

If NL tokens have been used in place of any ; s, the same transformation to schema conjunctions applies.

12.2.7.3 SchemaText

Given the above transformations of Declaration and DeclPart, any DeclPart in a SchemaText can be assumed to be a single expression.

A SchemaText with non-empty DeclPart and Predicate is equivalent to the schema construction expression containing that schema text.

$$e \mid p \implies [e \mid p]$$

If both DeclPart and Predicate are omitted, the schema text is equivalent to the set containing the empty binding.

$$\implies \{(\!|\,|\!)\}$$

If just the DeclPart is omitted, the schema text is equivalent to the schema construction expression in which there is a set containing the empty binding.

$$\mid p \implies [\{(\!|\,|\!)\} \mid p]$$

12.2.8 Name

These syntactic transformation rules address the concrete syntax productions DeclName, RefName, and OpName.

All DeclNames and RefNames that are NAMEs (not operator names) are transformed to those underlying NAMEs. Thus, any DeclName and RefName nodes in the parse tree are elided.

All operator names are transformed to NAMEs, by removing spaces and replacing each _ by a Z character that is not acceptable in concrete NAMEs. The Z character $\bowtie$ is used for this purpose here. The resulting NAME is given the same STROKEs as the component names of the operator, all of which shall have the same STROKEs.

Each resulting NAME shall be one for which there is an operator template paragraph in scope.

NOTE This excludes names made up of words from different operator templates.

EXAMPLE Given the operator templates

generic 30 leftassoc $(\_\ a\ \_\ b\ \_)$
generic 40 leftassoc $(\_\ c\ \_\ d\ \_)$

the following declaration conforms to the syntax but is excluded by this restriction.

$X\ a\ Y\ d\ Z == X \times Y \times Z$

In every operator name generated by syntactic transformation, for every '↘' WORDGLUE character in its WORD part, there shall be a paired following '↖' WORDGLUE character, for every '↗' WORDGLUE character in its WORD part, there shall be a paired following '↙' WORDGLUE character, and these shall occur only in nested pairs.

12.2.8.1 PrefixName

$$
\begin{aligned}
pre\ \_ &\Longrightarrow pre\bowtie \\
prep\ \_ &\Longrightarrow prep\bowtie \\
ln\ \_\ ess_1\ \ldots\ \_\ ess_{n-2}\ \_\ ere\ \_ &\Longrightarrow ln\bowtie ess_1 \ldots \bowtie ess_{n-2}\bowtie ere\bowtie \\
ln\ \_\ ess_1\ \ldots\ \_\ ess_{n-2}\ \_\ sre\ \_ &\Longrightarrow ln\bowtie ess_1 \ldots \bowtie ess_{n-2}\bowtie sre\bowtie \\
lp\ \_\ ess_1\ \ldots\ \_\ ess_{n-2}\ \_\ erep\ \_ &\Longrightarrow lp\bowtie ess_1 \ldots \bowtie ess_{n-2}\bowtie erep\bowtie \\
lp\ \_\ ess_1\ \ldots\ \_\ ess_{n-2}\ \_\ srep\ \_ &\Longrightarrow lp\bowtie ess_1 \ldots \bowtie ess_{n-2}\bowtie srep\bowtie
\end{aligned}
$$

12.2.8.2 PostfixName

$$
\begin{aligned}
\_\ post &\Longrightarrow \bowtie post \\
\_\ postp &\Longrightarrow \bowtie postp \\
\_\ el\ \_\ ess_2\ \ldots\ \_\ ess_{n-1}\ \_\ er &\Longrightarrow \bowtie el\bowtie ess_2 \ldots \bowtie ess_{n-1}\bowtie er \\
\_\ el\ \_\ ess_2\ \ldots\ \_\ ess_{n-1}\ \_\ sr &\Longrightarrow \bowtie el\bowtie ess_2 \ldots \bowtie ess_{n-1}\bowtie sr \\
\_\ elp\ \_\ ess_2\ \ldots\ \_\ ess_{n-1}\ \_\ erp &\Longrightarrow \bowtie elp\bowtie ess_2 \ldots \bowtie ess_{n-1}\bowtie erp \\
\_\ elp\ \_\ ess_2\ \ldots\ \_\ ess_{n-1}\ \_\ srp &\Longrightarrow \bowtie elp\bowtie ess_2 \ldots \bowtie ess_{n-1}\bowtie srp
\end{aligned}
$$

12.2.8.3 InfixName

$$
\begin{aligned}
\_\ in\ \_ &\Longrightarrow \bowtie in\bowtie \\
\_\ ip\ \_ &\Longrightarrow \bowtie ip\bowtie \\
\_\ el\ \_\ ess_2\ \ldots\ \_\ ess_{n-2}\ \_\ ere\ \_ &\Longrightarrow \bowtie el\bowtie ess_2 \ldots \bowtie ess_{n-2}\bowtie ere\bowtie \\
\_\ el\ \_\ ess_2\ \ldots\ \_\ ess_{n-2}\ \_\ sre\ \_ &\Longrightarrow \bowtie el\bowtie ess_2 \ldots \bowtie ess_{n-2}\bowtie sre\bowtie \\
\_\ elp\ \_\ ess_2\ \ldots\ \_\ ess_{n-2}\ \_\ erep\ \_ &\Longrightarrow \bowtie elp\bowtie ess_2 \ldots \bowtie ess_{n-2}\bowtie erep\bowtie \\
\_\ elp\ \_\ ess_2\ \ldots\ \_\ ess_{n-2}\ \_\ srep\ \_ &\Longrightarrow \bowtie elp\bowtie ess_2 \ldots \bowtie ess_{n-2}\bowtie srep\bowtie
\end{aligned}
$$

12.2.8.4 NofixName

$$
\begin{aligned}
ln\ \_\ ess_1\ \ldots\ \_\ ess_{n-1}\ \_\ er &\Longrightarrow ln\bowtie ess_1 \ldots \bowtie ess_{n-1}\bowtie er \\
ln\ \_\ ess_1\ \ldots\ \_\ ess_{n-1}\ \_\ sr &\Longrightarrow ln\bowtie ess_1 \ldots \bowtie ess_{n-1}\bowtie sr \\
lp\ \_\ ess_1\ \ldots\ \_\ ess_{n-1}\ \_\ erp &\Longrightarrow ln\bowtie ess_1 \ldots \bowtie ess_{n-1}\bowtie erp \\
lp\ \_\ ess_1\ \ldots\ \_\ ess_{n-1}\ \_\ srp &\Longrightarrow ln\bowtie ess_1 \ldots \bowtie ess_{n-1}\bowtie srp
\end{aligned}
$$

12.2.9 Generic name

All generic names are transformed to juxtapositions of NAMEs and generic parameter lists. This causes the generic operator definition paragraphs in which they appear to become generic horizontal definition paragraphs, and thus be amenable to further syntactic transformation.

Each resulting NAME shall be one for which there is an operator template paragraph in scope (see 12.2.8).

12.2.9.1 PrefixGenName

$$
\begin{aligned}
pre\ i &\Longrightarrow pre{\bowtie}\ [i] \\
ln\ i_1\ ess_1\ ...\ i_{n-2}\ ess_{n-2}\ i_{n-1}\ ere\ i_n &\Longrightarrow ln{\bowtie}ess_1...{\bowtie}ess_{n-2}{\bowtie}ere{\bowtie}\ [i_1, ..., i_{n-2}, i_{n-1}, i_n] \\
ln\ i_1\ ess_1\ ...\ i_{n-2}\ ess_{n-2}\ i_{n-1}\ sre\ i_n &\Longrightarrow ln{\bowtie}ess_1...{\bowtie}ess_{n-2}{\bowtie}sre{\bowtie}\ [i_1, ..., i_{n-2}, i_{n-1}, i_n]
\end{aligned}
$$

12.2.9.2 PostfixGenName

$$
\begin{aligned}
i\ post &\Longrightarrow {\bowtie}post\ [i] \\
i_1 el\ i_2\ ess_2\ ...\ i_{n-1}\ ess_{n-1}\ i_n\ er &\Longrightarrow {\bowtie}el{\bowtie}ess_2...{\bowtie}ess_{n-1}{\bowtie}er\ [i_1, i_2, ..., i_{n-1}, i_n] \\
i_1 el\ i_2\ ess_2\ ...\ i_{n-1}\ ess_{n-1}\ i_n\ sr &\Longrightarrow {\bowtie}el{\bowtie}ess_2...{\bowtie}ess_{n-1}{\bowtie}sr\ [i_1, i_2, ..., i_{n-1}, i_n]
\end{aligned}
$$

12.2.9.3 InfixGenName

$$
\begin{aligned}
i_1 in\ i_2 &\Longrightarrow {\bowtie}in{\bowtie}\ [i_1, i_2] \\
i_1 el\ i_2\ ess_2\ ...\ i_{n-2}\ ess_{n-2}\ i_{n-1}\ ere\ i_n &\Longrightarrow {\bowtie}el{\bowtie}ess_2...{\bowtie}ess_{n-2}{\bowtie}ere{\bowtie}\ [i_1, i_2, ..., i_{n-2}, i_{n-1}, i_n] \\
i_1 el\ i_2\ ess_2\ ...\ i_{n-2}\ ess_{n-2}\ i_{n-1}\ sre\ i_n &\Longrightarrow {\bowtie}el{\bowtie}ess_2...{\bowtie}ess_{n-2}{\bowtie}sre{\bowtie}\ [i_1, i_2, ..., i_{n-2}, i_{n-1}, i_n]
\end{aligned}
$$

12.2.9.4 NofixGenName

$$
\begin{aligned}
ln\ i_1\ ess_1\ ...\ i_{n-1}\ ess_{n-1}\ i_n\ er &\Longrightarrow ln{\bowtie}ess_1...{\bowtie}ess_{n-1}{\bowtie}er\ [i_1, ..., i_{n-1}, i_n] \\
ln\ i_1\ ess_1\ ...\ i_{n-1}\ ess_{n-1}\ i_n\ sr &\Longrightarrow ln{\bowtie}ess_1...{\bowtie}ess_{n-1}{\bowtie}sr\ [i_1, ..., i_{n-1}, i_n]
\end{aligned}
$$

12.2.10 Relation operator application

All relation operator applications are transformed to annotated membership predicates.

Each relation `NAME` shall be one for which there is an operator template paragraph in scope (see 12.2.8).

The left-hand sides of many of these transformation rules involve `ExpSep` phrases: they use *es* metavariables. None of them use *ss* metavariables: in other words, only the `Expression ES` case of `ExpSep` is specified, not the `ExpressionList SS` case. Where the latter case occurs in a specification, the `ExpressionList` shall be transformed by the rule in 12.2.12 to an expression, and thence a transformation analogous to that specified for the former case can be performed, differing only in that a *ss* appears in the relation name in place of an *es*.

12.2.10.1 PrefixRel

$$
\begin{aligned}
prep\ e &\Longrightarrow e \in prep{\bowtie} \\
lp\ e_1\ es_1\ ...\ e_{n-2}\ es_{n-2}\ e_{n-1}\ erep\ e_n &\Longrightarrow (e_1, ..., e_{n-2}, e_{n-1}, e_n) \in lp{\bowtie}es_1...{\bowtie}es_{n-2}{\bowtie}erep{\bowtie} \\
lp\ e_1\ es_1\ ...\ e_{n-2}\ es_{n-2}\ a_{n-1}\ srep\ e_n &\Longrightarrow (e_1, ..., e_{n-2}, a_{n-1}, e_n) \in lp{\bowtie}es_1...{\bowtie}es_{n-2}{\bowtie}srep{\bowtie}
\end{aligned}
$$

12.2.10.2 PostfixRel

$$
\begin{array}{rcl}
e\ postp & \Longrightarrow & e \in \bowtie postp \\
e_1\ elp\ e_2\ es_2\ ...\ e_{n-1}\ es_{n-1}\ e_n\ erp & \Longrightarrow & (e_1, e_2, ..., e_{n-1}, e_n) \in \bowtie elp \bowtie es_2 ... \bowtie es_{n-1} \bowtie erp \\
e_1\ elp\ e_2\ es_2\ ...\ e_{n-1}\ es_{n-1}\ a_n\ srp & \Longrightarrow & (e_1, e_2, ..., e_{n-1}, a_n) \in \bowtie elp \bowtie es_2 ... \bowtie es_{n-1} \bowtie srp
\end{array}
$$

12.2.10.3 InfixRel

$$
e_1\ ip_1\ e_2\ ip_2\ e_3\ ... \quad \Longrightarrow \quad e_1\ ip_1\ (e_2\ ⨾\ \alpha_1) \wedge (e_2\ ⨾\ \alpha_1)\ ip_2\ (e_3\ ⨾\ \alpha_2) ...
$$

The chained relation $e_1\ ip_1\ e_2\ ip_2\ e_3\ ...$ is semantically equivalent to a conjunction of relational predicates, with the constraint that duplicated expressions be of the same type.

$$
\begin{array}{rcl}
e_1 = e_2 & \Longrightarrow & e_1 \in \{e_2\} \\
e_1\ ip\ e_2 & \Longrightarrow & (e_1, e_2) \in \bowtie ip \bowtie
\end{array}
$$

ip in the above transformation is excluded from being $\in$ or $=$, whereas $ip_1, ip_2, ...$ in the chained relation can be $\in$ or $=$.

$$
\begin{array}{rcl}
e_1\ elp\ e_2\ es_2\ ...\ e_{n-2}\ es_{n-2}\ e_{n-1}\ erep\ e_n & \Longrightarrow & (e_1, e_2, ..., e_{n-2}, e_{n-1}, e_n) \in \bowtie elp \bowtie es_2 ... \bowtie es_{n-2} \bowtie erep \bowtie \\
e_1\ elp\ e_2\ es_2\ ...\ e_{n-2}\ es_{n-2}\ a_{n-1}\ srep\ e_n & \Longrightarrow & (e_1, e_2, ..., e_{n-2}, a_{n-1}, e_n) \in \bowtie elp \bowtie es_2 ... \bowtie es_{n-2} \bowtie srep \bowtie
\end{array}
$$

12.2.10.4 NofixRel

$$
\begin{array}{rcl}
lp\ e_1\ es_1\ ...\ e_{n-1}\ es_{n-1}\ e_n\ erp & \Longrightarrow & (e_1, ..., e_{n-1}, e_n) \in lp \bowtie es_1 ... \bowtie es_{n-1} \bowtie erp \\
lp\ e_1\ es_1\ ...\ e_{n-1}\ es_{n-1}\ a_n\ srp & \Longrightarrow & (e_1, ..., e_{n-1}, a_n) \in lp \bowtie es_1 ... \bowtie es_{n-1} \bowtie srp
\end{array}
$$

12.2.11 Function and generic operator application

All function operator applications are transformed to annotated application expressions.

All generic operator applications are transformed to annotated generic instantiation expressions.

For any particular function or generic operator application, two potential transformations are specified below, both of which result in the same `NAME`. That `NAME` shall be one for which there is an operator template paragraph in scope (see 12.2.8). Which of the two transformations is appropriate is determined by that operator template's category: function or generic respectively.

The left-hand sides of many of these transformation rules involve `ExpSep` phrases: they use *es* metavariables. None of them use *ss* metavariables: in other words, only the `Expression ES` case of `ExpSep` is specified, not the `ExpressionList SS` case. Where the latter case occurs in a specification, the `ExpressionList` shall be transformed by the rule in 12.2.12 to an expression, and thence a transformation analogous to that specified for the former case can be performed, differing only in that a *ss* appears in the function or generic name in place of an *es*.

12.2.11.1 `PrefixApp`

Function cases

$$\begin{aligned} pre\ e &\Longrightarrow pre{\bowtie}\ e \\ ln\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ ere\ e_n &\Longrightarrow ln{\bowtie}es_1\ldots{\bowtie}es_{n-2}{\bowtie}ere{\bowtie}\ (e_1, \ldots, e_{n-2}, e_{n-1}, e_n) \\ ln\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ sre\ e_n &\Longrightarrow ln{\bowtie}es_1\ldots{\bowtie}es_{n-2}{\bowtie}sre{\bowtie}\ (e_1, \ldots, e_{n-2}, a_{n-1}, e_n) \end{aligned}$$

Generic cases

$$\begin{aligned} pre\ e &\Longrightarrow pre{\bowtie}\ [e] \\ ln\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ ere\ e_n &\Longrightarrow ln{\bowtie}es_1\ldots{\bowtie}es_{n-2}{\bowtie}ere{\bowtie}\ [e_1, \ldots, e_{n-2}, e_{n-1}, e_n] \\ ln\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ sre\ e_n &\Longrightarrow ln{\bowtie}es_1\ldots{\bowtie}es_{n-2}{\bowtie}sre{\bowtie}\ [e_1, \ldots, e_{n-2}, a_{n-1}, e_n] \end{aligned}$$

12.2.11.2 `PostfixApp`

Function cases

$$\begin{aligned} e\ post &\Longrightarrow {\bowtie}post\ e \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ e_n\ er &\Longrightarrow {\bowtie}el{\bowtie}es_2\ldots{\bowtie}es_{n-1}{\bowtie}er\ (e_1, e_2, \ldots, e_{n-1}, e_n) \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ a_n\ sr &\Longrightarrow {\bowtie}el{\bowtie}es_2\ldots{\bowtie}es_{n-1}{\bowtie}sr\ (e_1, e_2, \ldots, e_{n-1}, a_n) \end{aligned}$$

Generic cases

$$\begin{aligned} e\ post &\Longrightarrow {\bowtie}post\ [e] \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ e_n\ er &\Longrightarrow {\bowtie}el{\bowtie}es_2\ldots{\bowtie}es_{n-1}{\bowtie}er\ [e_1, e_2, \ldots, e_{n-1}, e_n] \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ a_n\ sr &\Longrightarrow {\bowtie}el{\bowtie}es_2\ldots{\bowtie}es_{n-1}{\bowtie}sr\ [e_1, e_2, \ldots, e_{n-1}, a_n] \end{aligned}$$

12.2.11.3 `InfixApp`

Function cases

$$\begin{aligned} e_1\ in\ e_2 &\Longrightarrow {\bowtie}in{\bowtie}\ (e_1, e_2) \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ ere\ e_n &\Longrightarrow {\bowtie}el{\bowtie}es_2\ldots{\bowtie}es_{n-2}{\bowtie}ere{\bowtie}\ (e_1, e_2, \ldots, e_{n-2}, e_{n-1}, e_n) \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ sre\ e_n &\Longrightarrow {\bowtie}el{\bowtie}es_2\ldots{\bowtie}es_{n-2}{\bowtie}sre{\bowtie}\ (e_1, e_2, \ldots, e_{n-2}, a_{n-1}, e_n) \end{aligned}$$

Generic cases

$$\begin{aligned} e_1\ in\ e_2 &\Longrightarrow {\bowtie}in{\bowtie}\ [e_1, e_2] \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ ere\ e_n &\Longrightarrow {\bowtie}el{\bowtie}es_2\ldots{\bowtie}es_{n-2}{\bowtie}ere{\bowtie}\ [e_1, e_2, \ldots, e_{n-2}, e_{n-1}, e_n] \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ sre\ e_n &\Longrightarrow {\bowtie}el{\bowtie}es_2\ldots{\bowtie}es_{n-2}{\bowtie}sre{\bowtie}\ [e_1, e_2, \ldots, e_{n-2}, a_{n-1}, e_n] \end{aligned}$$

12.2.11.4 `NofixApp`

Function cases

$$ln\ e_1\ es_1\ ...\ e_{n-1}\ es_{n-1}\ e_n\ er \implies ln \bowtie es_1 ... \bowtie es_{n-1} \bowtie er\ (e_1, ..., e_{n-1}, e_n)$$
$$ln\ e_1\ es_1\ ...\ e_{n-1}\ es_{n-1}\ a_n\ sr \implies ln \bowtie es_1 ... \bowtie es_{n-1} \bowtie sr\ (e_1, ..., e_{n-1}, a_n)$$

Generic cases

$$ln\ e_1\ es_1\ ...\ e_{n-1}\ es_{n-1}\ e_n\ er \implies ln \bowtie es_1 ... \bowtie es_{n-1} \bowtie er\ [e_1, ..., e_{n-1}, e_n]$$
$$ln\ e_1\ es_1\ ...\ e_{n-1}\ es_{n-1}\ a_n\ sr \implies ln \bowtie es_1 ... \bowtie es_{n-1} \bowtie sr\ [e_1, ..., e_{n-1}, a_n]$$

12.2.12 Expression list

$$e_1, ..., e_n \implies \{(1, e_1), ..., (n, e_n)\}$$

Within an operator application, each expression list is syntactically transformed to the equivalent explicit representation of a sequence, which is a set of pairs of position and corresponding component expression.

13 Type inference rules

13.1 Introduction

All expressions in Z are typed, allowing some of the logical anomalies that can arise when sets are defined in terms of their properties to be avoided. An example of a Z phrase that is not well-typed is the predicate $2 \in 3$, because the second expression of a membership predicate is required to be a set of values, each of the same type as the first expression. The check for well-typedness of a Z specification can be automated, by conforming to the specification given in this clause.

The type constraints that shall be satisfied between the various parts of a Z phrase are specified by type inference rules, of which there is one corresponding to each annotated syntax production. The type inference rules together can be viewed as a partial function that maps a parse tree of an annotated syntax sentence to a fully annotated parse tree of an annotated syntax sentence.

Initially, all annotations are set to variables, and these are all distinct except as set by the syntactic transformation rules 12.2.7.1 and 12.2.10.3. A type inference rule's type sequent is a pattern that when matched against a phrase produces substitutions for its metavariables. Starting with a type sequent for a whole Z specification, that is matched against the pattern in the type inference rule for a sectioned specification. The resulting substitutions for metavariables are used to produce instantiations of the rule's type subsequents and side-conditions. The instantiated side-conditions include constraints that determine the environments to be used in typechecking the type subsequents. There is no need to solve the constraints yet. Instead, type inference rules can be applied to the generated type subsequents, each application producing zero or more new type subsequents, until no more type subsequents remain. This produces a tree of deductions, whose leaves correspond to the atomic phrases of the sentence, namely given types paragraphs, truth predicates, and reference expressions.

There remains a collection of constraints to be solved. There are dependencies between constraints: for example, a constraint that checks that a name is declared in an environment cannot be solved until that environment has been determined by other constraints. Unification is a suitable mechanism for solving constraints. A typechecker shall not impose any additional constraints, such as on the order in which constraints are expected to be solved [16].

For a well-typed specification, there shall be no contradictions amongst the constraints, and the solution to the constraints shall provide values for all of the variables. If there is a contradiction amongst the constraints, there can be no consistent assignment of annotations, and the specification is not well-typed. If the solution to the constraints does not provide a value for a variable, there is more than one possible assignment of annotations, and the specification is not well-typed.

EXAMPLE

$$\mid \; empty == \{\}$$

In this declaration, the type of *empty*, $\mathbb{P}\,\alpha$, involves an unconstrained variable.

13.2 Formal definition of type inference rules

13.2.1 Specification

13.2.1.1 Sectioned specification

$$\frac{\{\} \vdash^{\mathcal{S}} s_{prelude} \mathbin{:} \Gamma_0 \quad \delta_1 \vdash^{\mathcal{S}} s_1 \mathbin{:} \Gamma_1 \quad \ldots \quad \delta_n \vdash^{\mathcal{S}} s_n \mathbin{:} \Gamma_n}{\vdash^{\mathcal{Z}} s_1 \mathbin{:} \Gamma_1 \ldots s_n \mathbin{:} \Gamma_n} \left(\begin{array}{l} \delta_1 = \{prelude \mapsto \Gamma_0\} \\ \quad \vdots \\ \delta_n = \delta_{n-1} \cup \{i_{n-1} \mapsto \Gamma_{n-1}\} \end{array} \right)$$

where i_{n-1} is the name of section s_{n-1}, and none of the sections $s_1 \ldots s_n$ are named *prelude*.

Each section is typechecked in an environment formed from preceding sections, and is annotated with an environment that it establishes.

NOTE The section-type environment established by the prelude section is as follows.

$$\begin{array}{rcl} \Gamma_0 & = & (\mathbb{A}, (prelude, \mathbb{P}(\texttt{GIVEN}\ \mathbb{A}))); \\ & & (\mathbb{N}, (prelude, \mathbb{P}(\texttt{GIVEN}\ \mathbb{A}))); \\ & & (number\_literal\_0, (prelude, (\texttt{GIVEN}\ \mathbb{A}))); \\ & & (number\_literal\_1, (prelude, (\texttt{GIVEN}\ \mathbb{A}))); \\ & & (\bowtie\!+\!\bowtie, (prelude, \mathbb{P}(((\texttt{GIVEN}\ \mathbb{A}) \times (\texttt{GIVEN}\ \mathbb{A})) \times (\texttt{GIVEN}\ \mathbb{A})))) \end{array}$$

If one of the sections $s_1 \ldots s_n$ is named *prelude*, then the same type inference rule applies except that the type subsequent for the prelude section is omitted.

13.2.2 Section

13.2.2.1 Inheriting section

$$\frac{\beta_0 \vdash^{\mathcal{D}} D_1 \mathbin{:} \sigma_1 \quad \ldots \quad \beta_{n-1} \vdash^{\mathcal{D}} D_n \mathbin{:} \sigma_n}{\begin{array}{l} \Lambda \vdash^{\mathcal{S}} \texttt{ZED section}\ i \\ \qquad \texttt{parents}\ i_1, \ldots, i_m\ \texttt{END} \\ \quad D_1 \mathbin{:} \sigma_1 \ldots D_n \mathbin{:} \sigma_n \mathbin{:} \Gamma \end{array}} \left(\begin{array}{l} i \notin dom\ \Lambda \\ \{i_1, \ldots, i_m\} \subseteq dom\ \Lambda \\ \gamma_{-1} = \text{if}\ i = prelude\ \text{then}\ \{\}\ \text{else}\ \Lambda\ prelude \\ \gamma_0 = \gamma_{-1} \cup \Lambda\ i_1 \cup \ldots \cup \Lambda\ i_m \\ \beta_0 = \gamma_0 \fatsemi second \\ disjoint\ \langle dom\ \sigma_1, \ldots, dom\ \sigma_n \rangle \\ \Gamma \in (\_ \pfun \_) \\ \Gamma = \gamma_0 \cup \{j : \texttt{NAME};\ \tau : \texttt{Type} \mid j \mapsto \tau \in \sigma_1 \cup \ldots \cup \sigma_n \bullet j \mapsto (i, \tau)\} \\ \beta_1 = \beta_0 \cup \sigma_1 \\ \quad \vdots \\ \beta_{n-1} = \beta_{n-2} \cup \sigma_{n-1} \end{array} \right)$$

Taking the side-conditions in order, this type inference rule ensures that:

a) the name of the section, i, is different from that of any previous section;

b) the names in the parents list are names of known sections;

c) the section environment of the prelude is included if the section is not itself the prelude;

d) the section-type environment γ_o is formed from those of the parents;

e) the type environment β_o is determined from the section-type environment γ_o;

f) there is no global redefinition between any pair of paragraphs of the section (the sets of names in their signatures are disjoint);

g) a name which is common to the environments of multiple parents shall have originated in a common ancestral section, and a name introduced by a paragraph of this section shall not also be introduced by another paragraph or parent section (all ensured by the combined environment being a function);

h) the annotation of the section is an environment formed from those of its parents extended according to the signatures of its paragraphs;

i) and the type environment in which a paragraph is typechecked is formed from that of the parent sections extended with the signatures of the preceding paragraphs of this section.

NOTE 1 Ancestors need not be immediate parents, and a section cannot be amongst its own ancestors (no cycles in the parent relation).

NOTE 2 The name of a section can be the same as the name of a variable introduced in a declaration—the two are not confused.

13.2.3 Generic instantiation

Generic declarations can appear only at the paragraph level. The types of generic declarations shall be determined before the constraints arising from the side-conditions of the type inference rules for references to generics can be solved (see 13.2.6.1 and 13.2.6.2). The constraints for each paragraph shall have a unique solution without any consideration of the constraints of following paragraphs. That is, the constraints shall be solved in per-paragraph batches. Having determined the types of references to generic declarations, instantiations that were left implicit are made explicit, ready for subsequent semantic relation.

NOTE This is why generic instantiation is defined here, immediately before the type inference rules for paragraphs.

13.2.3.1 Generic type instantiation

The constraints that cannot be solved until the type of a generic declaration is determined are those that involve the operation of generic type instantiation. The generic type instantiation meta-function relates a known generic type and a list of argument types to the type in which each reference to a generic parameter has been substituted with the corresponding argument type. Applications of the generic type instantiation meta-function are formulated here as the juxtaposition of a generic type (parenthesized) with a square-bracketed list of argument types.

$$
\begin{aligned}
([i_1, ..., i_n]\ \texttt{GIVEN}\ i)\ [\tau_1, ..., \tau_n] &= \texttt{GIVEN}\ i \\
([i_1, ..., i_n]\ \texttt{GENTYPE}\ i_k)\ [\tau_1, ..., \tau_n] &= \tau_k \\
([i_1, ..., i_n]\ \mathbb{P}\,\tau)\ [\tau_1, ..., \tau_n] &= \mathbb{P}(([i_1, ..., i_n]\ \tau)\ [\tau_1, ..., \tau_n]) \\
([i_1, ..., i_n]\ \tau'_1 \times ... \times \tau'_m)\ [\tau_1, ..., \tau_n] &= ([i_1, ..., i_n]\ \tau'_1)\ [\tau_1, ..., \tau_n] \times ... \times ([i_1, ..., i_n]\ \tau'_m)\ [\tau_1, ..., \tau_n] \\
([i_1, ..., i_n]\ [i'_1 : \tau'_1;\ ...;\ i'_m : \tau'_m])\ [\tau_1, ..., \tau_n] & \\
&= [i'_1 : [i_1, ..., i_n]\ \tau'_1\ [\tau_1, ..., \tau_n];\ ...;\ i'_m : [i_1, ..., i_n]\ \tau'_m\ [\tau_1, ..., \tau_n]]
\end{aligned}
$$

13.2.3.2 Carrier set

The meta-function *carrier* relates a type phrase to an expression phrase denoting the carrier set of that type. It is used for the calculation of implicit generic actuals, and also later in semantic transformation rules.

$$
\begin{aligned}
carrier\ (\mathtt{GIVEN}\ i) &= i\heartsuit \mathbin{:\!\!\in} \mathbb{P}(\mathtt{GIVEN}\ i) \\
carrier\ (\mathtt{GENTYPE}\ i) &= i\spadesuit \mathbin{:\!\!\in} \mathbb{P}(\mathtt{GENTYPE}\ i) \\
carrier\ (\mathbb{P}\,\tau) &= \mathbb{P}(carrier\ \tau) \mathbin{:\!\!\in} \mathbb{P}\,\mathbb{P}\,\tau \\
carrier\ (\tau_1 \times ... \times \tau_n) &= (carrier\ \tau_1 \times ... \times carrier\ \tau_n) \mathbin{:\!\!\in} \mathbb{P}(\tau_1 \times ... \times \tau_n) \\
carrier\ ([i_n : \tau_n;\ ...;\ i_n : \tau_n]) &= [i_n : carrier\ \tau_n;\ ...;\ i_n : carrier\ \tau_n] \mathbin{:\!\!\in} \mathbb{P}[i_n : \tau_n;\ ...;\ i_n : \tau_n]
\end{aligned}
$$

NOTE 1 The expressions are generated with type annotations, to avoid needing to apply type inference again, and so avoid the potential problem of type names being captured by local declarations.

NOTE 2 But for the GIVEN/GENTYPE distinction, the ♡ and ♠ strokes and the generation of type annotations, each of these equations generates an expression that has the same textual appearance as the type.

NOTE 3 There is no equation for generic types because they appear in only the type annotation of generic axiomatic paragraphs, and *carrier* is never applied there.

13.2.3.3 Implicit instantiation

The value of a reference expression that refers to a generic definition is an inferred instantiation of that generic definition.

$$
i \mathbin{:\!\!\in} [i_1,\ ...,\ i_n]\tau, \tau' \overset{\tau' = ([i_1,\ ...,\ i_n]\tau)}{\Longrightarrow} [\alpha_1,\ ...,\ \alpha_n]\ \ i\ [carrier\ \alpha_1,\ ...,\ carrier\ \alpha_n] \mathbin{:\!\!\in} \tau'
$$

It is semantically equivalent to the generic instantiation expression whose generic actuals are the carrier sets of the types inferred for the generic parameters. The type τ' is an instantiation of the generic type τ. The types inferred for the generic parameters are α_1, ..., α_n. They shall all be determinable by comparison of τ with τ' as suggested by the condition on the transformation. Cases where these types cannot be so determined, because the generic type is independent of some of the generic parameters, are not well-typed.

EXAMPLE 1 The paragraph

$a[X] == 1$

defines a with type $[X]\mathtt{GIVEN}\ \mathbb{A}$. The paragraph

$b == a$

typechecks, giving the annotated expression $a \mathbin{:\!\!\in} [X]\mathtt{GIVEN}\ \mathbb{A}, \mathtt{GIVEN}\ \mathbb{A}$. Comparison of the generic type with the instantiated type does not determine a type for the generic parameter X, and so this specification is not well-typed.

Cases where these types are not unique (contain unconstrained variables) are not well-typed.

EXAMPLE 2 The paragraph

$empty == \varnothing$

will contain the annotated expression $\varnothing \mathbin{:\!\!\in} [X]\,\mathbb{P}\,X, \mathbb{P}\,\alpha$, in which the type determined for the generic parameter X is unconstrained, and so this specification is not well-typed.

13.2.4 Paragraph

13.2.4.1 Given types paragraph

$$
\frac{}{\Sigma \vdash^{\mathcal{D}} \mathtt{ZED}\ [i_1, ..., i_n]\ \mathtt{END} \mathbin{:\!\!\in} \sigma} \left(\begin{array}{l} \#\{i_1,\ ...,\ i_n\} = n \\ \sigma = i_1 : \mathbb{P}(\mathtt{GIVEN}\ i_1);\ ...;\ i_n : \mathbb{P}(\mathtt{GIVEN}\ i_n) \end{array} \right)
$$

In a given types paragraph, there shall be no duplication of names. The annotation of the paragraph is a signature associating the given type names with powerset types.

13.2.4.2 Axiomatic description paragraph

$$
\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:\!\!\in} \tau}{\Sigma \vdash^{\mathcal{D}} \mathtt{AX}\ e \mathbin{:\!\!\in} \tau\ \mathtt{END} \mathbin{:\!\!\in} \sigma} (\tau = \mathbb{P}[\sigma])
$$

In an axiomatic description paragraph AX e END, the expression e shall be a schema. The annotation of the paragraph is the signature of that schema.

13.2.4.3 Generic axiomatic description paragraph

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_n)\} \vdash^{\mathcal{E}} e \ \S\ \tau}{\Sigma \vdash^{\mathcal{D}} \texttt{GENAX}\ [i_1, ..., i_n]\ e\ \S\ \tau\ \texttt{END}\ \S\ \sigma} \left(\begin{array}{l} \#\{i_1,\ ...,\ i_n\} = n \\ \tau = \mathbb{P}[\beta] \\ \sigma = \lambda\ j : dom\ \beta \bullet [i_1, ..., i_n]\ (\beta\ j) \end{array} \right)$$

In a generic axiomatic description paragraph $\texttt{GENAX}\ [i_1, ..., i_n]\ e\ \texttt{END}$, there shall be no duplication of names within the generic parameters. The expression e is typechecked, in an environment overridden by the generic parameters, and shall be a schema. The annotation of the paragraph is formed from the signature of that schema, having the same names but associated with types that are generic.

13.2.4.4 Free types paragraph

$$\frac{\begin{array}{ccc} \beta \vdash^{\mathcal{E}} e_{1\,1} \ \S\ \tau_{1\,1} & ... & \beta \vdash^{\mathcal{E}} e_{1\,n_1} \ \S\ \tau_{1\,n_1} \\ \vdots & & \\ \beta \vdash^{\mathcal{E}} e_{r\,1} \ \S\ \tau_{r\,1} & ... & \beta \vdash^{\mathcal{E}} e_{r\,n_r} \ \S\ \tau_{r\,n_r} \end{array}}{\Sigma \vdash^{\mathcal{D}} \texttt{ZED} \quad \begin{array}{l} f_1 ::= h_{1\,1} \mid ... \mid h_{1\,m_1} \mid \\ \quad g_{1\,1} \langle\!\langle e_{1\,1} \ \S\ \tau_{1\,1} \rangle\!\rangle \mid \\ \quad \vdots \mid \\ \quad g_{1\,n_1} \langle\!\langle e_{1\,n_1} \ \S\ \tau_{1\,n_1} \rangle\!\rangle \ \& \\ \vdots\ \& \\ f_r ::= h_{r\,1} \mid ... \mid h_{r\,m_r} \mid \\ \quad g_{r\,1} \langle\!\langle e_{r\,1} \ \S\ \tau_{r\,1} \rangle\!\rangle \mid \\ \quad \vdots \mid \\ \quad g_{r\,n_r} \langle\!\langle e_{r\,n_r} \ \S\ \tau_{r\,n_r} \rangle\!\rangle \end{array} \texttt{END}\ \S\ \sigma} \left(\begin{array}{l} \#\{f_1,\ h_{1\,1},\ ...,\ h_{1\,m_1},\ g_{1\,1},\ ...,\ g_{1\,n_1}, \\ \vdots, \\ \quad f_r,\ h_{r\,1},\ ...,\ h_{r\,m_r},\ g_{r\,1},\ ...,\ g_{r\,n_r}\} \\ = r + m_1 + ... + m_r + n_1 + ... + n_r \\ \beta = \Sigma \oplus \{f_1 \mapsto \mathbb{P}(\texttt{GIVEN}\ f_1),\ ...,\ f_r \mapsto \mathbb{P}(\texttt{GIVEN}\ f_r)\} \\ \tau_{1\,1} = \mathbb{P}\,\alpha_{1\,1} \quad ... \quad \tau_{1\,n_1} = \mathbb{P}\,\alpha_{1\,n_1} \\ \quad \vdots \\ \tau_{r\,1} = \mathbb{P}\,\alpha_{r\,1} \quad ... \quad \tau_{r\,n_r} = \mathbb{P}\,\alpha_{r\,n_r} \\ \sigma = f_1 : \mathbb{P}(\texttt{GIVEN}\ f_1); \\ \quad h_{1\,1} : \texttt{GIVEN}\ f_1;\ ...;\ h_{1\,m_1} : \texttt{GIVEN}\ f_1; \\ \quad g_{1\,1} : \mathbb{P}(\tau_{1\,1} \times \texttt{GIVEN}\ f_1); \\ \quad\quad \vdots; \\ \quad g_{1\,n_1} : \mathbb{P}(\tau_{1\,n_1} \times \texttt{GIVEN}\ f_1); \\ \quad \vdots; \\ f_r : \mathbb{P}(\texttt{GIVEN}\ f_r); \\ \quad h_{r\,1} : \texttt{GIVEN}\ f_r;\ ...;\ h_{r\,m_r} : \texttt{GIVEN}\ f_r; \\ \quad g_{r\,1} : \mathbb{P}(\tau_{r\,1} \times \texttt{GIVEN}\ f_r); \\ \quad\quad \vdots; \\ \quad g_{r\,n_r} : \mathbb{P}(\tau_{r\,n_r} \times \texttt{GIVEN}\ f_r) \end{array} \right)$$

In a free types paragraph, the names of the free types, elements and injections shall all be different. The expressions representing the domains of the injections are typechecked in an environment overridden by the names of the free types, and shall all be sets. The annotation of the paragraph is the signature whose names are those of all the free types, the elements, and the injections, each associated with the corresponding type.

13.2.4.5 Conjecture paragraph

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \texttt{ZED} \vdash ?\ p\ \texttt{END}\ \S\ \sigma} (\sigma = \epsilon)$$

In a conjecture paragraph $\texttt{ZED} \vdash ?\ p\ \texttt{END}$, the predicate p shall be well-typed. The annotation of the paragraph is the empty signature.

13.2.4.6 Generic conjecture paragraph

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_n)\} \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \texttt{ZED}\ [i_1, ..., i_n] \vdash ?\ p\ \texttt{END}\ \S\ \sigma} \left(\begin{array}{l} \#\{i_1,\ ...,\ i_n\} = n \\ \sigma = \epsilon \end{array} \right)$$

In a generic conjecture paragraph $\texttt{ZED}\ [i_1, ..., i_n] \vdash ?\ p\ \texttt{END}$, there shall be no duplication of names within the generic parameters. The predicate p shall be well-typed in an environment overridden by the generic parameters. The annotation of the paragraph is the empty signature.

13.2.5 Predicate

13.2.5.1 Membership predicate

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{:} \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathbin{:} \tau_2}{\Sigma \vdash^{\mathcal{P}} (e_1 \mathbin{:} \tau_1) \in (e_2 \mathbin{:} \tau_2)} (\ \tau_2 = \mathbb{P}\, \tau_1\ )$$

In a membership predicate $e_1 \in e_2$, expression e_2 shall be a set, and expression e_1 shall be of the same type as the members of set e_2.

13.2.5.2 Truth predicate

$$\frac{}{\Sigma \vdash^{\mathcal{P}} \mathrm{true}}$$

A truth predicate is always well-typed.

13.2.5.3 Negation predicate

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \neg\, p}$$

A negation predicate $\neg\, p$ is well-typed if and only if predicate p is well-typed.

13.2.5.4 Conjunction predicate

$$\frac{\Sigma \vdash^{\mathcal{P}} p_1 \qquad \Sigma \vdash^{\mathcal{P}} p_2}{\Sigma \vdash^{\mathcal{P}} p_1 \land p_2}$$

A conjunction predicate $p_1 \land p_2$ is well-typed if and only if predicates p_1 and p_2 are well-typed.

13.2.5.5 Universal quantification predicate

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \tau \qquad \Sigma \oplus \beta \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \forall (e \mathbin{:} \tau) \bullet p} (\tau = \mathbb{P}[\beta])$$

In a universal quantification predicate $\forall\, e \bullet p$, expression e shall be a schema, and predicate p shall be well-typed in the environment overridden by the signature of schema e.

13.2.5.6 Unique existential quantification predicate

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \tau \qquad \Sigma \oplus \beta \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \exists_1 (e \mathbin{:} \tau) \bullet p} (\tau = \mathbb{P}[\beta])$$

In a unique existential quantification predicate $\exists_1\ e \bullet p$, expression e shall be a schema, and predicate p shall be well-typed in the environment overridden by the signature of schema e.

13.2.6 Expression

13.2.6.1 Reference expression

In a reference expression, if the name is of the form Δi and no declaration of this name yet appears in the environment, then the following syntactic transformation is applied.

$$\Delta i \overset{\Delta i \notin dom\ \Sigma}{\Longrightarrow} [i;\ i\ ']$$

This syntactic transformation makes the otherwise undefined name be equivalent to the corresponding schema construction expression, which is then typechecked.

Similarly, if the name is of the form Ξi and no declaration of this name yet appears in the environment, then the following syntactic transformation is applied.

$$\Xi i \overset{\Xi i \notin dom\ \Sigma}{\Longrightarrow} [i;\ i\ ' \mid \theta\, i = \theta\, i\ ']$$

NOTE 1 The Ξ notation is deliberately not defined in terms of the Δ notation.

NOTE 2 Type inference could be done without these syntactic transformations, but they are necessary steps in defining the formal semantics.

NOTE 3 Only occurrences of Δ and Ξ that are in such reference expressions are so transformed, not others such as those in the names of declarations.

$$\frac{}{\Sigma \vdash^{\varepsilon} i \ \wr \ \tau} \left(\begin{array}{l} i \in dom\ \Sigma \\ \tau = \text{if } generic\_type\ (\Sigma\ i) \text{ then } \Sigma\ i, (\Sigma\ i)\ [\alpha_1, ..., \alpha_n] \text{ else } \Sigma\ i \end{array} \right)$$

In any other reference expression i, the name i shall be associated with a type in the environment. If that type is generic, the annotation of the whole expression is a pair of both the uninstantiated type (for the Instantiation clause to determine that this is a reference to a generic definition) and the type instantiated with new distinct type variables (which the context shall constrain to non-generic types). Otherwise (if the type in the environment is non-generic), that is the type of the whole expression.

NOTE 4 If the type is generic, the reference expression will be transformed to a generic instantiation expression by the rule in 13.2.3.3. That shall be done only when the implicit instantiations have been determined via constraints on the new type variables $\alpha_1, ..., \alpha_n$.

13.2.6.2 Generic instantiation expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ \wr \ \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \ \wr \ \tau_n}{\Sigma \vdash^{\varepsilon} i[(e_1 \ \wr \ \tau_1), ..., (e_n \ \wr \ \tau_n)] \ \wr \ \tau} \left(\begin{array}{l} i \in dom\ \Sigma \\ generic\_type\ (\Sigma\ i) \\ \tau_1 = \mathbb{P}\,\alpha_1 \\ \quad \vdots \\ \tau_n = \mathbb{P}\,\alpha_n \\ \tau = (\Sigma\ i)\ [\alpha_1, ..., \alpha_n] \end{array} \right)$$

In a generic instantiation expression $i\ [e_1, ..., e_n]$, the name i shall be associated with a generic type in the environment, and the expressions $e_1, ..., e_n$ shall be sets. That generic type shall have the same number of parameters as there are sets. The type of the whole expression is the instantiation of that generic type by the types of those sets' components.

NOTE The operation of generic type instantiation is defined in 13.2.3.1.

13.2.6.3 Set extension expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ \wr \ \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \ \wr \ \tau_n}{\Sigma \vdash^{\varepsilon} \{(e_1 \ \wr \ \tau_1), ..., (e_n \ \wr \ \tau_n)\} \ \wr \ \tau} \left(\begin{array}{l} \text{if } n > 0 \text{ then} \\ \quad (\tau_1 = \tau_n \\ \quad\quad \vdots \\ \quad \tau_{n-1} = \tau_n \\ \quad \tau = \mathbb{P}\,\tau_1) \\ \text{else } \tau = \mathbb{P}\,\alpha \end{array} \right)$$

In a set extension expression, every component expression shall be of the same type. The type of the whole expression is a powerset of the components' type, or a powerset of a type variable if there are no components. In the latter case, the variable shall be constrained by the context, otherwise the specification is not well-typed.

13.2.6.4 Set comprehension expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ \wr \ \tau_1 \qquad \Sigma \oplus \beta \vdash^{\varepsilon} e_2 \ \wr \ \tau_2}{\Sigma \vdash^{\varepsilon} \{(e_1 \ \wr \ \tau_1) \bullet (e_2 \ \wr \ \tau_2)\} \ \wr \ \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_3 = \mathbb{P}\,\tau_2 \end{array} \right)$$

In a set comprehension expression $\{e_1 \bullet e_2\}$, expression e_1 shall be a schema. The type of the whole expression is a powerset of the type of expression e_2, as determined in an environment overridden by the signature of schema e_1.

13.2.6.5 Powerset expression

$$\frac{\Sigma \vdash^{\varepsilon} e \ : \ \tau_1}{\Sigma \vdash^{\varepsilon} \mathbb{P}(e \ : \ \tau_1) \ : \ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}\,\alpha \\ \tau_2 = \mathbb{P}\,\tau_1 \end{array} \right)$$

In a powerset expression $\mathbb{P}\,e$, expression e shall be a set. The type of the whole expression is then a powerset of the type of expression e.

13.2.6.6 Tuple extension expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ : \ \tau_1 \quad \ldots \quad \Sigma \vdash^{\varepsilon} e_n \ : \ \tau_n}{\Sigma \vdash^{\varepsilon} ((e_1 \ : \ \tau_1), \ldots, (e_n \ : \ \tau_n)) \ : \ \tau} (\ \tau = \tau_1 \times \ldots \times \tau_n \ )$$

In a tuple extension expression $(e_1, ..., e_n)$, the type of the whole expression is the Cartesian product of the types of the individual component expressions.

13.2.6.7 Tuple selection expression

$$\frac{\Sigma \vdash^{\varepsilon} e \ : \ \tau_1}{\Sigma \vdash^{\varepsilon} (e \ : \ \tau_1) \,.\, b \ : \ \tau_2} \left(\begin{array}{l} \tau_1 = \alpha_1 \times \ldots \times \alpha_k \\ b \in 1 \,..\, k \\ \tau_2 = \alpha_b \end{array} \right)$$

In a tuple selection expression $e \,.\, b$, the type of expression e shall be a Cartesian product, and the numeric value of NUMERAL b shall select one of its components. The type of the whole expression is the type of the selected component.

13.2.6.8 Binding extension expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ : \ \tau_1 \quad \ldots \quad \Sigma \vdash^{\varepsilon} e_n \ : \ \tau_n}{\Sigma \vdash^{\varepsilon} \langle\!| \ i_1 == (e_1 \ : \ \tau_1), \ldots, i_n == (e_n \ : \ \tau_n) \ |\!\rangle \ : \ \tau} \left(\begin{array}{l} \# \{i_1, \ \ldots, \ i_n\} = n \\ \tau = [i_1 : \tau_1; \ \ldots; \ i_n : \tau_n] \end{array} \right)$$

In a binding extension expression $\langle\!| \ i_1 == e_1, ..., i_n == e_n \ |\!\rangle$, there shall be no duplication amongst the bound names. The type of the whole expression is that of a binding whose signature associates the names with the types of the corresponding expressions.

13.2.6.9 Binding construction expression

$$\frac{\Sigma \vdash^{\varepsilon} e \ : \ \tau_1}{\Sigma \vdash^{\varepsilon} \theta\,(e \ : \ \tau_1)^{\,*} \ : \ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \forall \ i : dom \ \beta \bullet (i \ decor^{\,*}, \beta \ i) \in \Sigma \wedge \neg \ generic\_type \ (\beta \ i) \\ \tau_2 = [\beta] \end{array} \right)$$

In a binding construction expression $\theta\, e\ ^*$, the expression e shall be a schema. Every name and type pair in its signature, with the optional decoration added, shall be present in the environment with a non-generic type. The type of the whole expression is that of a binding whose signature is that of the schema.

NOTE If the type in the environment were generic, the semantic transformation in 14.2.5.2 would produce a reference expression whose implicit instantiation is not determined by this International Standard.

13.2.6.10 Binding selection expression

$$\frac{\Sigma \vdash^{\varepsilon} e \ : \ \tau_1}{\Sigma \vdash^{\varepsilon} (e \ : \ \tau_1) \,.\, i \ : \ \tau_2} \left(\begin{array}{l} \tau_1 = [\beta] \\ (i, \ \tau_2) \in \beta \end{array} \right)$$

In a binding selection expression $e \ . \ i$, expression e shall be a binding, and name i shall select one of its components. The type of the whole expression is the type of the selected component.

13.2.6.11 Application expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ : \ \tau_1 \quad \Sigma \vdash^{\varepsilon} e_2 \ : \ \tau_2}{\Sigma \vdash^{\varepsilon} (e_1 \ : \ \tau_1)\,(e_2 \ : \ \tau_2) \ : \ \tau_3} (\ \tau_1 = \mathbb{P}(\tau_2 \times \tau_3) \ )$$

In an application expression $e_1\ e_2$, the expression e_1 shall be a set of pairs, and expression e_2 shall be of the same type as the first components of those pairs. The type of the whole expression is the type of the second components of those pairs.

13.2.6.12 Definite description expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{:} \tau_1 \qquad \Sigma \oplus \beta \vdash^{\mathcal{E}} e_2 \mathbin{:} \tau_2}{\Sigma \vdash^{\mathcal{E}} \mu\,(e_1 \mathbin{:} \tau_1) \bullet (e_2 \mathbin{:} \tau_2) \mathbin{:} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_3 = \tau_2 \end{array} \right)$$

In a definite description expression $\mu\ e_1 \bullet e_2$, expression e_1 shall be a schema. The type of the whole expression is the type of expression e_2, as determined in an environment overridden by the signature of schema e_1.

13.2.6.13 Variable construction expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \tau_1}{\Sigma \vdash^{\mathcal{E}} [i : (e \mathbin{:} \tau_1)] \mathbin{:} \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}\,\alpha \\ \tau_2 = \mathbb{P}[i : \alpha] \end{array} \right)$$

In a variable construction expression $[i : e]$, expression e shall be a set. The type of the whole expression is that of a schema whose signature associates the name i with the type of a member of the set e.

13.2.6.14 Schema construction expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \tau_1 \qquad \Sigma \oplus \beta \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{E}} [(e \mathbin{:} \tau_1) \mid p] \mathbin{:} \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = \tau_1 \end{array} \right)$$

In a schema construction expression $[e \mid p]$, expression e shall be a schema, and predicate p shall be well-typed in an environment overridden by the signature of schema e. The type of the whole expression is the same as the type of expression e.

13.2.6.15 Schema negation expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \tau_1}{\Sigma \vdash^{\mathcal{E}} \neg\,(e \mathbin{:} \tau_1) \mathbin{:} \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = \tau_1 \end{array} \right)$$

In a schema negation expression $\neg\ e$, expression e shall be a schema. The type of the whole expression is the same as the type of expression e.

13.2.6.16 Schema conjunction expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{:} \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathbin{:} \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \mathbin{:} \tau_1) \wedge (e_2 \mathbin{:} \tau_2) \mathbin{:} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[\beta_1 \cup \beta_2] \end{array} \right)$$

In a schema conjunction expression $e_1 \wedge e_2$, expressions e_1 and e_2 shall be schemas, and their signatures shall be compatible. The type of the whole expression is that of the schema whose signature is the union of those of expressions e_1 and e_2.

13.2.6.17 Schema hiding expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \tau_1}{\Sigma \vdash^{\mathcal{E}} (e \mathbin{:} \tau_1) \setminus (i_1, \ldots, i_n) \mathbin{:} \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \{i_1, \ldots, i_n\} \subseteq dom\ \beta \\ \tau_2 = \mathbb{P}[\{i_1, \ldots, i_n\} \lhd \beta] \end{array} \right)$$

In a schema hiding expression $e \setminus (i_1, \ldots, i_n)$, expression e shall be a schema, and the names to be hidden shall all be in the signature of that schema. The type of the whole expression is that of a schema whose signature is computed by subtracting from the signature of expression e those pairs whose names are hidden.

13.2.6.18 Schema universal quantification expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \tau_1 \qquad \Sigma \oplus \beta_1 \vdash^{\varepsilon} e_2 \mathbin{:} \tau_2}{\Sigma \vdash^{\varepsilon} \forall (e_1 \mathbin{:} \tau_1) \bullet (e_2 \mathbin{:} \tau_2) \mathbin{:} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[dom\ \beta_1 \lhd \beta_2] \end{array} \right)$$

In a schema universal quantification expression $\forall e_1 \bullet e_2$, expression e_1 shall be a schema, and expression e_2, in an environment overridden by the signature of schema e_1, shall also be a schema, and the signatures of these two schemas shall be compatible. The type of the whole expression is that of a schema whose signature is computed by subtracting from the signature of e_2 those pairs whose names are in the signature of e_1.

13.2.6.19 Schema unique existential quantification expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \tau_1 \qquad \Sigma \oplus \beta_1 \vdash^{\varepsilon} e_2 \mathbin{:} \tau_2}{\Sigma \vdash^{\varepsilon} \exists_1 (e_1 \mathbin{:} \tau_1) \bullet (e_2 \mathbin{:} \tau_2) \mathbin{:} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[dom\ \beta_1 \lhd \beta_2] \end{array} \right)$$

In a schema unique existential quantification expression $\exists_1 e_1 \bullet e_2$, expression e_1 shall be a schema, and expression e_2, in an environment overridden by the signature of schema e_1, shall also be a schema, and the signatures of these two schemas shall be compatible. The type of the whole expression is that of a schema whose signature is computed by subtracting from the signature of e_2 those pairs whose names are in the signature of e_1.

13.2.6.20 Schema renaming expression

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{:} \tau_1}{\Sigma \vdash^{\varepsilon} (e \mathbin{:} \tau_1)[j_1 \,/\, i_1, ..., j_n \,/\, i_n] \mathbin{:} \tau_2} \left(\begin{array}{l} \#\{i_1, ..., i_n\} = n \\ \tau_1 = \mathbb{P}[\beta_1] \\ \beta_2 = \{j_1 \mapsto i_1, ..., j_n \mapsto i_n\} \fatsemi \beta_1 \cup \{i_1, ..., i_n\} \lhd \beta_1 \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_2 \in (\_ \mapsto \_) \end{array} \right)$$

In a schema renaming expression $e\ [j_1 \,/\, i_1, ..., j_n \,/\, i_n]$, there shall be no duplicates amongst the old names $i_1, ..., i_n$. Expression e shall be a schema. The type of the whole expression is that of a schema whose signature is like that of expression e but with the new names in place of corresponding old names. Declarations that are merged by the renaming shall have the same type.

NOTE Old names need not be in the signature of the schema. This is so as to permit renaming to distribute over other notations such as disjunction.

13.2.6.21 Schema precondition expression

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{:} \tau_1}{\Sigma \vdash^{\varepsilon} \mathrm{pre}\ (e \mathbin{:} \tau_1) \mathbin{:} \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = \mathbb{P}[\{i, j : \mathtt{NAME} \mid j \in dom\ \beta \wedge (j = i\ decor\ ' \vee j = i\ decor\ !) \bullet j\} \lhd \beta] \end{array} \right)$$

In a schema precondition expression $\mathrm{pre}\, e$, expression e shall be a schema. The type of the whole expression is that of a schema whose signature is computed by subtracting from the signature of e those pairs whose names have primed or shrieked decorations.

13.2.6.22 Schema composition expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \tau_1 \qquad \Sigma \vdash^{\varepsilon} e_2 \mathbin{:} \tau_2}{\Sigma \vdash^{\varepsilon} (e_1 \mathbin{:} \tau_1) \fatsemi (e_2 \mathbin{:} \tau_2) \mathbin{:} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ match = \{i : dom\ \beta_2 \mid i\ decor\ ' \in dom\ \beta_1 \bullet i\} \\ \beta_3 = \{i : match \bullet i\ decor\ '\} \lhd \beta_1 \\ \beta_4 = match \lhd \beta_2 \\ \beta_3 \approx \beta_4 \\ \{i : match \bullet i \mapsto \beta_1(i\ decor\ ')\} \approx \{i : match \bullet i \mapsto \beta_2\ i\} \\ \tau_3 = \mathbb{P}[\beta_3 \cup \beta_4] \end{array} \right)$$

In a schema composition expression $e_1 \fatsemi e_2$, expressions e_1 and e_2 shall be schemas. Let *match* be the set of names in schema e_2 for which there are matching primed names in schema e_1. Let β_3 be the signature formed from the components of e_1 excluding the matched primed components. Let β_4 be the signature formed from the components of e_2 excluding the matched unprimed components. Signatures β_3 and β_4 shall be compatible. The types of the excluded matched pairs of components shall be the same. The type of the whole expression is that of a schema whose signature is the union of β_3 and β_4.

13.2.6.23 Schema piping expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 : \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 : \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 : \tau_1) \gg (e_2 : \tau_2) : \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ match = \{i : \mathtt{NAME} \mid i\ decor\ ! \in dom\ \beta_1 \land i\ decor\ ? \in dom\ \beta_2 \bullet i\} \\ \beta_3 = \{i : match \bullet i\ decor\ !\} \lhd \beta_1 \\ \beta_4 = \{i : match \bullet i\ decor\ ?\} \lhd \beta_2 \\ \beta_3 \approx \beta_4 \\ \{i : match \bullet i \mapsto \beta_1 (i\ decor\ !)\} \approx \{i : match \bullet i \mapsto \beta_2\ (i\ decor\ ?)\} \\ \tau_3 = \mathbb{P}[\beta_3 \cup \beta_4] \end{array} \right)$$

In a schema piping expression $e_1 \gg e_2$, expressions e_1 and e_2 shall be schemas. Let *match* be the set of names for which there are matching shrieked names in schema e_1 and queried names in schema e_2. Let β_3 be the signature formed from the components of e_1 excluding the matched shrieked components. Let β_4 be the signature formed from the components of e_2 excluding the matched queried components. Signatures β_3 and β_4 shall be compatible. The types of the excluded matched pairs of components shall be the same. The type of the whole expression is that of a schema whose signature is the union of β_3 and β_4.

13.2.6.24 Schema decoration expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e : \tau_1}{\Sigma \vdash^{\mathcal{E}} (e : \tau_1)^{+} : \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = \mathbb{P}[\{i : dom\ \beta \bullet i\ decor^{+} \mapsto \beta\ i\}] \end{array} \right)$$

In a schema decoration expression e^{+}, expression e shall be a schema. The type of the whole expression is that of a schema whose signature is like that of e but with the stroke appended to each of its names.

13.3 Summary of scope rules

NOTE Here is an informal explanation of the scope rules implied by the type inference rules in 13.2.

A scope is static: it depends on only the structure of the text, not on the value of any predicate or expression.

A declaration can be either: a given type, a free type, a formal generic parameter, or an instance of `Declaration` usually within a `DeclPart`.

The scopes of given types and free types (which occur only at paragraph level), and `Declaration`s at paragraph level (such as those of schema definitions and those of the outermost `DeclPart` in axiomatic descriptions), are the whole of the rest of the section and any sections of which that is an ancestor.

Redeclaration at paragraph level of any name already declared at paragraph level is prohibited. Redeclaration at an inner level of any name already declared with larger scope makes a hole in the scope of the outer declaration.

In a free types paragraph, the scopes of the declarations of the free types include the right-hand sides of the free type declarations, whereas the scopes of the declarations of the elements and injections of the free types do not include the free types paragraph itself.

The scope of a formal generic parameter is the rest of the paragraph in which it appears.

A `DeclPart` is not in the scope of its declarations.

The declarations of a schema inclusion declaration are distinct from those in the signature of the schema itself, and

so have separate scopes.

A name may be declared more than once within a `DeclPart` provided the types of the several declarations are identical. In this case, the declarations are merged, so that they share the same scope, and the corresponding properties are conjoined.

The scope of the declarations in the `DeclPart` of a quantification, set comprehension, function construction, definite description or schema construction expression is the | part of the `SchemaText` and any • part of that construct.

14 Semantic transformation rules

14.1 Introduction

The semantic transformation rules define some annotated notations as being equivalent to other annotated notations. The only sentences of concern here are ones that are already known to be well-formed syntactically and well-typed. These semantic transformations are transformations that could not appear earlier as syntactic transformations because they depend on type annotations or generic instantiations or are applicable only to parse trees of phrases that are not in the concrete syntax.

Some semantic transformation rules generate other transformable notation, though exhaustive application of these rules always terminates. They introduce no type errors. It is not intended that type inference be repeated on the generated notation, though type annotations are needed on that notation for the semantic relations. Nevertheless, the manipulation of type annotations is not made explicit throughout these rules, as that would be obfuscatory and can easily be derived by the reader. Indeed, some rules exploit concrete notation for brevity and clarity.

The semantic transformation rules are listed in the same order as the corresponding productions of the annotated syntax.

All applications of transformation rules that generate new declarations shall choose the names of those declarations to be such that they do not capture references.

14.2 Formal definition of semantic transformation rules

14.2.1 Specification

There are no semantic transformation rules for specifications.

14.2.2 Section

There are no semantic transformation rules for sections.

14.2.3 Paragraph

14.2.3.1 Free types paragraph

A free types paragraph is semantically equivalent to the sequence of given type paragraph and axiomatic definition paragraph defined here.

NOTE 1 This exploits notation that is not present in the annotated syntax for the purpose of abbreviation.

ZED
$f_1 ::= h_{1\,1} \mid \ldots \mid h_{1\,m_1} \mid g_{1\,1} \langle\langle e_{1\,1} \rangle\rangle \mid \ldots \mid g_{1\,n_1} \langle\langle e_{1\,n_1} \rangle\rangle$
& ... &
$f_r ::= h_{r\,1} \mid \ldots \mid h_{r\,m_r} \mid g_{r\,1} \langle\langle e_{r\,1} \rangle\rangle \mid \ldots \mid g_{r\,n_r} \langle\langle e_{r\,n_r} \rangle\rangle$
END

$\Longrightarrow$

ZED
$[f_1, \ldots, f_r]$
END

$$
\begin{array}{l}
\texttt{AX} \\
h_{1\,1},\ ...,\ h_{1\,m_1} : f_1 \\
\vdots \\
h_{r\,1},\ ...,\ h_{r\,m_r} : f_r \\
g_{1\,1} : \mathbb{P}(e_{1\,1} \times f_1);\ ...;\ g_{1\,n_1} : \mathbb{P}(e_{1\,n_1} \times f_1) \\
\vdots \\
g_{r\,1} : \mathbb{P}(e_{r\,1} \times f_r);\ ...;\ g_{r\,n_r} : \mathbb{P}(e_{r\,n_r} \times f_r) \\
| \\
(\forall\ u : e_{1\,1} \bullet \exists_1\ x : g_{1\,1} \bullet x\ .\ 1 = u) \land ... \land (\forall\ u : e_{1\,n_1} \bullet \exists_1\ x : g_{1\,n_1} \bullet x\ .\ 1 = u) \\
\vdots \land \\
(\forall\ u : e_{r\,1} \bullet \exists_1\ x : g_{r\,1} \bullet x\ .\ 1 = u) \land ... \land (\forall\ u : e_{r\,n_r} \bullet \exists_1\ x : g_{r\,n_r} \bullet x\ .\ 1 = u) \\
\\
(\forall\ u, v : e_{1\,1} \mid g_{1\,1} u = g_{1\,1} v \bullet u = v) \land ... \land (\forall\ u, v : e_{1\,n_1} \mid g_{1\,n_1} u = g_{1\,n_1} v \bullet u = v) \\
\vdots \land \\
(\forall\ u, v : e_{r\,1} \mid g_{r\,1} u = g_{r\,1} v \bullet u = v) \land ... \land (\forall\ u, v : e_{r\,n_r} \mid g_{r\,n_r} u = g_{r\,n_r} v \bullet u = v) \\
\\
\forall\ b_1, b_2 : \mathbb{N} \bullet \\
\quad (\forall\ w : f_1 \mid \\
\qquad (b_1 = 1 \land w = h_{1\,1} \lor ... \lor b_1 = m_1 \land w = h_{1\,m_1} \lor \\
\qquad\quad b_1 = m_1 + 1 \land w \in \{x : g_{1\,1} \bullet x\ .\ 2\} \lor ... \lor b_1 = m_1 + n_1 \land w \in \{x : g_{1\,n_1} \bullet x\ .\ 2\}) \\
\qquad \land (b_2 = 1 \land w = h_{1\,1} \lor ... \lor b_2 = m_1 \land w = h_{1\,m_1} \lor \\
\qquad\quad b_2 = m_1 + 1 \land w \in \{x : g_{1\,1} \bullet x\ .\ 2\} \lor ... \lor b_2 = m_1 + n_1 \land w \in \{x : g_{1\,n_1} \bullet x\ .\ 2\}) \bullet \\
\qquad\quad b_1 = b_2) \land \\
\quad \vdots \land \\
\quad (\forall\ w : f_r \mid \\
\qquad (b_1 = 1 \land w = h_{r\,1} \lor ... \lor b_1 = m_r \land w = h_{r\,m_r} \lor \\
\qquad\quad b_1 = m_r + 1 \land w \in \{x : g_{r\,1} \bullet x\ .\ 2\} \lor ... \lor b_1 = m_r + n_r \land w \in \{x : g_{r\,n_r} \bullet x\ .\ 2\}) \\
\qquad \land (b_2 = 1 \land w = h_{r\,1} \lor ... \lor b_2 = m_r \land w = h_{r\,m_r} \lor \\
\qquad\quad b_2 = m_r + 1 \land w \in \{x : g_{r\,1} \bullet x\ .\ 2\} \lor ... \lor b_2 = m_r + n_r \land w \in \{x : g_{r\,n_r} \bullet x\ .\ 2\}) \bullet \\
\qquad\quad b_1 = b_2) \\
\\
\forall\ w_1 : \mathbb{P}\, f_1;\ ...;\ w_r : \mathbb{P}\, f_r \mid \\
\qquad h_{1\,1} \in w_1 \land ... \land h_{1\,m_1} \in w_1 \land \\
\qquad \vdots \land \\
\qquad h_{r\,1} \in w_r \land ... \land h_{r\,m_r} \in w_r \land \\
\qquad (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{1\,1}) \bullet g_{1\,1} y \in w_1) \land \\
\qquad ... \land (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{1\,n_1}) \bullet g_{1\,n_1} y \in w_1) \land \\
\qquad \vdots \land \\
\qquad (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{r\,1}) \bullet g_{r\,1} y \in w_r) \land \\
\qquad ... \land (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{r\,n_r}) \bullet g_{r\,n_r} y \in w_r) \bullet \\
\quad w_1 = f_1 \land ... \land w_r = f_r \\
\texttt{END}
\end{array}
$$

The type names are introduced by the given types paragraph. The elements are declared as members of their corresponding free types. The injections are declared as functions from values in their domains to their corresponding free type.

The first of the four blank-line separated predicates is the total functionality property. It ensures that for every injection, the injection is functional at every value in its domain.

The second of the four blank-line separated predicates is the injectivity property. It ensures that for every injection, any pair of values in its domain for which the injection returns the same value shall be a pair of equal values (hence the name injection).

The third of the four blank-line separated predicates is the disjointness property. It ensures that for every free type, every pair of values of the free type are equal only if they are the same element or are returned by application of the same injection to equal values.

The fourth of the four blank-line separated predicates is the induction property. It ensures that for every free type, its members are its elements, the values returned by its injections, and nothing else.

The generated μ expressions in the induction property are intended to effect substitutions of all references to the free type names, including any such references within generic instantiation lists in the e expressions.

NOTE 2 That is why this is a semantic transformation not a syntactic one: all implicit generic instantiations shall have been made explicit before it is applied.

NOTE 3 The right-hand side of this transformation could have been expressed using notation from the mathematical toolkit, as follows, but for the desire to define the Z core language separately from the mathematical toolkit.

ZED
$[f_1, ..., f_r]$
END

AX
$h_{1\,1},\ ...,\ h_{1\,m_1} : f_1$
$\vdots$
$h_{r\,1},\ ...,\ h_{r\,m_r} : f_r$
$g_{1\,1} : e_{1\,1} \rightarrowtail f_1;\ ...;\ g_{1\,n_1} : e_{1\,n_1} \rightarrowtail f_1$
$\vdots$
$g_{r\,1} : e_{r\,1} \rightarrowtail f_r;\ ...;\ g_{r\,n_r} : e_{r\,n_r} \rightarrowtail f_r$
$|$
$disjoint\langle \{h_{1\,1}\},\ ...,\ \{h_{1\,m_1}\},\ ran\, g_{1\,1},\ ...,\ ran\, g_{1\,n_1} \rangle$
$\vdots$
$disjoint\langle \{h_{r\,1}\},\ ...,\ \{h_{r\,m_r}\},\ ran\, g_{r\,1},\ ...,\ ran\, g_{r\,n_r} \rangle$
$\forall\ w_1 : \mathbb{P}\, f_1;\ ...;\ w_r : \mathbb{P}\, f_r \mid$
$\quad \{h_{1\,1},\ ...,\ h_{1\,m_1}\} \cup g_{1\,1}(\!|\ \mu\, f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{1\,1}\ |\!)$
$\quad\quad \cup ... \cup g_{1\,n_1}(\!|\ \mu\, f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{1\,n_1}\ |\!) \subseteq w_1 \land$
$\quad \vdots \land$
$\quad \{h_{r\,1},\ ...,\ h_{r\,m_r}\} \cup g_{r\,1}(\!|\ \mu\, f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{r\,1}\ |\!)$
$\quad\quad \cup ... \cup g_{r\,n_r}(\!|\ \mu\, f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{r\,n_r}\ |\!) \subseteq w_r \bullet$
$\quad w_1 = f_1 \land ... \land w_r = f_r$
END

14.2.4 Predicate

14.2.4.1 Unique existential predicate

The unique existential quantification predicate $\exists_1\ e \bullet p$ is *true* if and only if there is exactly one value for e for which p is *true*.

$$\exists_1\ e \bullet p \quad \Longrightarrow \quad \neg\ (\forall\ e \bullet \neg\ (p \land (\forall\ [e \mid p]^{\bowtie} \bullet \theta\, e = \theta\, e^{\bowtie})))$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$\exists_1\ e \bullet p \quad \Longrightarrow \quad \exists\ e \bullet p \wedge (\forall\ [e \mid p]^{\bowtie} \bullet \theta\, e = \theta\, e^{\bowtie})$$

It is semantically equivalent to there existing at least one value for e for which p is *true* and all those values for which it is *true* being the same.

14.2.5 Expression

14.2.5.1 Tuple selection expression

The value of the tuple selection expression $e\ .\ b$ is the b'th component of the tuple that is the value of e.

$$(e\ \mathbin{:}\ \tau_1 \times ... \times \tau_n)\ .\ b \quad \Longrightarrow \quad \{i : carrier\ (\tau_1 \times ... \times \tau_n) \bullet (i, \mu\ i_1 : carrier\ \tau_1;\ ...;\ i_n : carrier\ \tau_n \mid i = (i_1, ..., i_n) \bullet i_b)\}\ e$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$(e\ \mathbin{:}\ \tau_1 \times ... \times \tau_n)\ .\ b \quad \Longrightarrow \quad (\lambda\ i : carrier\ (\tau_1 \times ... \times \tau_n) \bullet \mu\ i_1 : carrier\ \tau_1;\ ...;\ i_n : carrier\ \tau_n \mid i = (i_1, ..., i_n) \bullet i_b)\ e$$

It is semantically equivalent to the function construction, from tuples of the Cartesian product type to the selected component of the tuple b, applied to the particular tuple e.

14.2.5.2 Binding construction expression

The value of the binding construction expression $\theta\ e\ ^*$ is the binding whose names are those in the signature of schema e and whose values are those of the same names with the optional decoration appended.

$$\theta\ e\ ^*\ \mathbin{:}\ [i_1 : \tau_1;\ ...;\ i_n : \tau_n] \quad \Longrightarrow \quad \langle\!|\ i_1 == i_1\ decor\ ^*, ..., i_n == i_n\ decor\ ^*\ |\!\rangle$$

It is semantically equivalent to the binding extension expression whose value is that binding.

14.2.5.3 Binding selection expression

The value of the binding selection expression $e\ .\ i$ is that value associated with i in the binding that is the value of e.

$$(e\ \mathbin{:}\ [\sigma])\ .\ i \quad \Longrightarrow \quad \{carrier\ [\sigma] \bullet (chartuple\ (carrier\ \ [\sigma]), i)\}\ e$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$(e\ \mathbin{:}\ [\sigma])\ .\ i \quad \Longrightarrow \quad (\lambda\ carrier\ [\sigma] \bullet i)\ e$$

It is semantically equivalent to the function construction expression, from bindings of the schema type of e, to the value of the selected name i, applied to the particular binding e.

14.2.5.4 Application expression

The value of the application expression $e_1\ e_2$ is the sole value associated with e_2 in the relation e_1.

$$e_1\ e_2\ \mathbin{:}\ \tau \quad \Longrightarrow \quad (\mu\ i : carrier\ \tau \mid (e_2, i) \in e_1 \bullet i)$$

It is semantically equivalent to that sole range value i such that the pair (e_2, i) is in the set of pairs that is the value of e_1. If there is no value or more than one value associated with e_2, then the application expression has a value but what it is is not specified.

14.2.5.5 Schema hiding expression

The value of the schema hiding expression $e \setminus (i_1, ..., i_n)$ is that schema whose signature is that of schema e minus the hidden names, and whose bindings have the same values as those in schema e.

$$(e \ ⨟ \ \mathbb{P}[\sigma]) \setminus (i_1, ..., i_n) \quad \Longrightarrow \quad \neg \ (\forall \ i_1 : carrier \ (\sigma \ i_1); \ ...; \ i_n : carrier \ (\sigma \ i_n) \bullet \neg \ e)$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$(e \ ⨟ \ \mathbb{P}[\sigma]) \setminus (i_1, ..., i_n) \quad \Longrightarrow \quad \exists \ i_1 : carrier \ (\sigma \ i_1); \ ...; \ i_n : carrier \ (\sigma \ i_n) \bullet e$$

It is semantically equivalent to the schema existential quantification of the hidden names $i_1, ..., i_n$ from the schema e.

14.2.5.6 Schema unique existential quantification expression

The value of the schema unique existential quantification expression $\exists_1 \ e_1 \bullet e_2$ is the set of bindings of schema e_2 restricted to exclude names that are in the signature of e_1, for at least one binding of the schema e_1.

$$\exists_1 \ e_1 \bullet e_2 \quad \Longrightarrow \quad \neg \ (\forall \ e_1 \bullet \neg \ (e_2 \wedge (\forall \ [e_1 \mid e_2]^{\bowtie} \bullet \theta \ e_1 = \theta \ e_1 \ ^{\bowtie})))$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$\exists_1 \ e_1 \bullet e_2 \quad \Longrightarrow \quad \exists \ e_1 \bullet e_2 \wedge (\forall \ [e_1 \mid e_2]^{\bowtie} \bullet \theta \ e_1 = \theta \ e_1 \ ^{\bowtie})$$

It is semantically equivalent to a schema existential quantification expression, analogous to the unique existential quantification predicate transformation.

14.2.5.7 Schema precondition expression

The value of the schema precondition expression $\mathrm{pre}\, e$ is that schema which is like schema e but without its primed and shrieked components.

$$\mathrm{pre}(e \ ⨟ \ \mathbb{P}[\sigma_1]) \ ⨟ \ \mathbb{P}[\sigma_2] \quad \Longrightarrow \quad \neg \ (\forall \ carrier \ [\sigma_1 \setminus \sigma_2] \bullet \neg \ e)$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$\mathrm{pre}(e \ ⨟ \ \mathbb{P}[\sigma_1]) \ ⨟ \ \mathbb{P}[\sigma_2] \quad \Longrightarrow \quad \exists \ carrier \ [\sigma_1 \setminus \sigma_2] \bullet e$$

It is semantically equivalent to the existential quantification of the primed and shrieked components from the schema e.

14.2.5.8 Schema composition expression

The value of the schema composition expression $e_1 \fatsemi e_2$ is that schema representing the operation of doing the operations represented by schemas e_1 and e_2 in sequence.

$$(e_1 \ ⨟ \ \mathbb{P}[\sigma_1]) \fatsemi (e_2 \ ⨟ \ \mathbb{P}[\sigma_2]) \ ⨟ \ \mathbb{P}[\sigma]$$
$$\Longrightarrow$$
$$\begin{array}{l}\neg \ (\forall \ e^{\bowtie} \bullet \neg \ (\neg \ (\forall \ e_3 \bullet \neg \ [e_1; \ e^{\bowtie} \mid \theta \ e_3 = \theta \ e^{\bowtie}]) \\ \qquad \wedge \neg \ (\forall \ e_4 \bullet \neg \ [e_2; \ e^{\bowtie} \mid \theta \ e_4 = \theta \ e^{\bowtie}])))\end{array}$$
$$\begin{array}{l}\textit{where } e_3 == carrier \ [\{i : \mathtt{NAME}; \ \tau : \mathtt{Type} \mid i \ decor \ ' \mapsto \tau \in \sigma_1 \bullet i \ decor \ ' \mapsto \tau\}] \\ \quad \textit{and } e_4 == carrier \ [\{i : \mathtt{NAME}; \ \tau : \mathtt{Type} \mid i \mapsto \tau \in \sigma_2 \bullet i \mapsto \tau\}] \\ \quad \textit{and } e^{\bowtie} == (e_4)^{\bowtie}\end{array}$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$(e_1 \mathbin{:} \mathbb{P}[\sigma_1]) \fatsemi (e_2 \mathbin{:} \mathbb{P}[\sigma_2]) \mathbin{:} \mathbb{P}[\sigma]$$
$$\Longrightarrow$$
$$\begin{array}{l} \exists\, e^{\bowtie} \bullet (\exists\, e_3 \bullet [e_1;\ e^{\bowtie} \mid \theta\, e_3 = \theta\, e^{\bowtie}]) \\ \quad \land\ (\exists\, e_4 \bullet [e_2;\ e^{\bowtie} \mid \theta\, e_4 = \theta\, e^{\bowtie}]) \\ \mathit{where}\ e_3 == \mathit{carrier}\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i\ \mathit{decor}\ ' \mapsto \tau \in \sigma_1 \bullet i\ \mathit{decor}\ ' \mapsto \tau\}] \\ \quad \mathit{and}\ e_4 == \mathit{carrier}\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i \mapsto \tau \in \sigma_2 \bullet i \mapsto \tau\}] \\ \quad \mathit{and}\ e^{\bowtie} == (e_4)^{\bowtie} \end{array}$$

It is semantically equivalent to the existential quantification of the matched pairs of primed components of e_1 and unprimed components of e_2, with those matched pairs being equated.

14.2.5.9 Schema piping expression

The value of the schema piping expression $e_1 \gg e_2$ is that schema representing the operation formed from the two operations represented by schemas e_1 and e_2 with the outputs of e_1 identified with the inputs of e_2.

$$(e_1 \mathbin{:} \mathbb{P}[\sigma_1]) \gg (e_2 \mathbin{:} \mathbb{P}[\sigma_2]) \mathbin{:} \mathbb{P}[\sigma]$$
$$\Longrightarrow$$
$$\begin{array}{l} \neg\, (\forall\, e^{\bowtie} \bullet \neg\, (\neg\, (\forall\, e_3 \bullet \neg\, [e_1;\ e^{\bowtie} \mid \theta\, e_3 = \theta\, e^{\bowtie}]) \\ \qquad \land\ \neg\, (\forall\, e_4 \bullet \neg\, [e_2;\ e^{\bowtie} \mid \theta\, e_4 = \theta\, e^{\bowtie}]))) \\ \mathit{where}\ e_3 == \mathit{carrier}\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i\ \mathit{decor}\ ! \mapsto \tau \in \sigma_1 \bullet i\ \mathit{decor}\ ! \mapsto \tau\}] \\ \quad \mathit{and}\ e_4 == \mathit{carrier}\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i\ \mathit{decor}\ ? \mapsto \tau \in \sigma_2 \bullet i\ \mathit{decor}\ ? \mapsto \tau\}] \\ \quad \mathit{and}\ e^{\bowtie} == (e_4)^{\bowtie} \end{array}$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$(e_1 \mathbin{:} \mathbb{P}[\sigma_1]) \gg (e_2 \mathbin{:} \mathbb{P}[\sigma_2]) \mathbin{:} \mathbb{P}[\sigma]$$
$$\Longrightarrow$$
$$\begin{array}{l} \exists\, e^{\bowtie} \bullet (\exists\, e_3 \bullet [e_1;\ e^{\bowtie} \mid \theta\, e_3 = \theta\, e^{\bowtie}]) \\ \quad \land\ (\exists\, e_4 \bullet [e_2;\ e^{\bowtie} \mid \theta\, e_4 = \theta\, e^{\bowtie}]) \\ \mathit{where}\ e_3 == \mathit{carrier}\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i\ \mathit{decor}\ ! \mapsto \tau \in \sigma_1 \bullet i\ \mathit{decor}\ ! \mapsto \tau\}] \\ \quad \mathit{and}\ e_4 == \mathit{carrier}\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i\ \mathit{decor}\ ? \mapsto \tau \in \sigma_2 \bullet i\ \mathit{decor}\ ? \mapsto \tau\}] \\ \quad \mathit{and}\ e^{\bowtie} == (e_4)^{\bowtie} \end{array}$$

It is semantically equivalent to the existential quantification of the matched pairs of shrieked components of e_1 and queried components of e_2, with those matched pairs being equated.

14.2.5.10 Schema decoration expression

The value of the schema decoration expression $e\ ^{+}$ is that schema whose bindings are like those of the schema e except that their names have the addition stroke $^{+}$.

$$(e \mathbin{:} \mathbb{P}[i_1 : \tau_1;\ \ldots;\ i_n : \tau_n])^{+} \quad \Longrightarrow \quad e\ [i_1\ \mathit{decor}\ ^{+} / i_1, \ldots, i_n\ \mathit{decor}\ ^{+} / i_n]$$

It is semantically equivalent to the schema renaming where decorated names rename the original names.

15 Semantic relations

15.1 Introduction

The semantic relations define the meaning of the remaining annotated notation (that not defined by semantic transformation rules) by relation to sets of models in ZF set theory. The only sentences of concern here are ones that are already known to be well-formed syntactically and well-typed.

This clause defines the meaning of a Z specification in terms of the semantic values that its global variables may take consistent with the constraints imposed on them by the specification.

This definition is loose: it leaves the values of ill-formed definite description expressions undefined. It is otherwise tight: it specifies the values of all expressions that do not depend on values of ill-formed definite descriptions,

every predicate is either *true* or *false*, and every expression denotes a value. The looseness leaves the values of undefined expressions unspecified. Any particular semantics conforms to this International Standard if it is consistent with this loose definition.

EXAMPLE The predicate $(\mu\, x : \{\,\}) \in T$ could be either *true* or *false* depending on the treatment of undefinedness.

NOTE 1 Typical specifications contain expressions that in some circumstances have undefined values. In those circumstances, those expressions ought not to affect the meaning of the specification. This definition is then sufficiently tight.

NOTE 2 Alternative treatments of undefined expressions include one or more bottoms outside of the carrier sets, or undetermined values from within the carrier sets.

15.2 Formal definition of semantic relations

15.2.1 Specification

15.2.1.1 Sectioned specification

$$[\![\ s_1\ \ldots\ s_n\ ]\!]^{Z} \;=\; ([\![\ \texttt{ZED}\ \text{section}\ \mathit{prelude}\ \text{parents}\ \texttt{END}\ \ldots\ ]\!]^{S} \fatsemi [\![\ s_1\ ]\!]^{S} \fatsemi \ldots \fatsemi [\![\ s_n\ ]\!]^{S})\ \varnothing$$

The meaning of the Z specification s_1 ... s_n is the function from sections' names to their sets of models formed by starting with the empty function and extending that with a maplet from a section's name to its set of models for each section in the specification, starting with the prelude.

To determine $[\![\ \texttt{ZED}\ \text{section}\ \mathit{prelude}\ \text{parents}\ \texttt{END}\ \ldots\ ]\!]^{Z}$ another prelude shall not be prefixed onto it.

NOTE The meaning of a specification is not the meaning of its last section, so as to permit several meaningful units within a single document.

15.2.2 Section

15.2.2.1 Inheriting section

The prelude section, as defined in clause 11, is treated specially, as it is the only one that does not have prelude as an implicit parent.

$$[\![\ \texttt{ZED}\ \text{section}\ \mathit{prelude}\ \text{parents}\ \texttt{END}\ D_1\ \ldots\ D_n\ ]\!]^{S} = \lambda\, \mathcal{T} : \mathit{SectionModels} \bullet \{\mathit{prelude} \mapsto ([\![\ D_1\ ]\!]^{\mathcal{D}} \fatsemi \ldots \fatsemi [\![\ D_n\ ]\!]^{\mathcal{D}})\ (\!|\ \{\varnothing\}\ |\!)\}$$

The meaning of the prelude section is given by that constant function which, whatever function from sections' names and their sets of models it is given, returns the singleton set mapping the name *prelude* to its set of models. The set of models is that to which the set containing an empty model is related by the composition of the relations between models that denote the meanings of each of the prelude's paragraphs—see clause 11 for details of those paragraphs.

NOTE One model of the prelude section can be written as follows.

$\{\mathbb{A} \mapsto \mathbb{A},$
$\mathbb{N} \mapsto \mathbb{N},$
$\mathit{number\_literal\_0} \mapsto 0,$
$\mathit{number\_literal\_1} \mapsto 1,$
$\_ + \_ \mapsto \{((0,0),0),((0,1),1),((1,0),1),((1,1),2),\ldots\}\}$

The behaviour of $(\_ + \_)$ on non-natural numbers, e.g. reals, has not been defined at this point, so the set of models for the prelude section includes alternatives for every possible extended behaviour of addition.

$$[\![\texttt{ZED section } i \texttt{ parents } i_1, ..., i_m \texttt{ END } D_1 \ ... \ D_n]\!]^{\mathcal{S}} = \lambda\ T : SectionModels \bullet T \cup \{i \mapsto (([\![D_1]\!]^{\mathcal{D}} \fatsemi ... \fatsemi [\![D_n]\!]^{\mathcal{D}}) (\!| \{M_0 : T\ prelude;\ M_1 : T\ i_1;\ ...;\ M_m : T\ i_m;\ M : Model \mid M = M_0 \cup M_1 \cup ... \cup M_m \bullet M\} |\!)\}$$

The meaning of a section other than the prelude is the extension of a function from sections' names to their sets of models with a maplet from the given section's name to its set of models. The given section's set of models is that to which the union of the models of the section's parents is related by the composition of the relations between models that denote the meanings of each of the section's paragraphs.

15.2.3 Paragraph

15.2.3.1 Given types paragraph

The given types paragraph ZED $[i_1, ..., i_n]$ END introduces unconstrained global names.

$$[\![\texttt{ZED } [i_1, ..., i_n] \texttt{ END}]\!]^{\mathcal{D}} = \{M : Model;\ w_1, ..., w_n : \mathbb{W} \bullet M \mapsto M \cup \{i_1 \mapsto w_1, ..., i_n \mapsto w_n\} \cup \{i_1\ decor\ \heartsuit \mapsto w_1, ..., i_n\ decor\ \heartsuit \mapsto w_n\}\}$$

It relates a model M to that model extended with associations between the names of the given types and semantic values chosen to represent their carrier sets. Associations for names decorated with the reserved stroke $\heartsuit$ are also introduced, so that references to them from given types (see 15.2.6.1) can avoid being captured.

15.2.3.2 Axiomatic description paragraph

The axiomatic description paragraph AX e END introduces global names and constraints on their values.

$$[\![\texttt{AX } e \texttt{ END}]\!]^{\mathcal{D}} = \{M : Model;\ t : \mathbb{W} \mid t \in [\![e]\!]^{\mathcal{E}} M \bullet M \mapsto M \cup t\}$$

It relates a model M to that model extended with a binding t of the schema that is the value of e in model M.

15.2.3.3 Generic axiomatic description paragraph

The generic axiomatic description paragraph GENAX $[i_1, ..., i_n]$ e END introduces global names and constraints on their values, with generic parameters that have to be instantiated (by sets) whenever those names are referenced.

$$[\![\texttt{GENAX } [i_1, ..., i_n] \ (e \ \fatsemi \ \mathbb{P}[j_1 : \tau_1; \ ...; \ j_m : \tau_m]) \texttt{ END}]\!]^{\mathcal{D}} = \{M : Model;\ u : \mathbb{W} \upharpoonright n \rightarrow \mathbb{W} \mid \forall\ w_1, ..., w_n : \mathbb{W} \bullet \exists\ w : \mathbb{W} \bullet u\ (w_1, ..., w_n) \in w \land (M \oplus \{i_1 \mapsto w_1, ..., i_n \mapsto w_n\} \cup \{i_1\ decor\ \spadesuit \mapsto w_1, ..., i_n\ decor\ \spadesuit \mapsto w_n\}) \mapsto w \in [\![e]\!]^{\mathcal{E}} \bullet M \mapsto M \cup \lambda\ y : \{j_1, ..., j_m\} \bullet \lambda\ x : \mathbb{W} \upharpoonright n \bullet u\ x\ y\}$$

Given a model M and generic argument sets w_1, ..., w_n, the semantic value of the schema e in that model overridden by the association of the generic parameter names with those sets is w. All combinations of generic argument sets are considered. The function u maps the generic argument sets to a binding in the schema w. The paragraph relates the model M to that model extended with the binding that associates the names of the schema e (namely j_1, ..., j_m) with the corresponding value in the binding resulting from application of u to arbitrary instantiating sets x. Associations for names decorated with the reserved stroke $\spadesuit$ are also introduced whilst determining the semantic value of e, so that references to them from generic types (see 15.2.6.2) can avoid being captured.

15.2.3.4 Conjecture paragraph

The conjecture paragraph ZED $\vdash ?\ p$ END expresses a property that may logically follow from the specification. It may be a starting point for a proof.

$$[\![\texttt{ZED} \vdash ?\ p \texttt{ END}]\!]^{\mathcal{D}} = id\ Model$$

It relates a model to itself: the truth of p in a model does not affect the meaning of the specification.

15.2.3.5 Generic conjecture paragraph

The generic conjecture paragraph ZED $[i_1, ..., i_n] \vdash ?\ p$ END expresses a generic property that may logically follow from the specification. It may be a starting point for a proof.

$$[\![\texttt{ZED}\ [i_1, ..., i_n] \vdash ?\ p\ \texttt{END}]\!]^{\mathcal{P}} \quad = \quad id\ Model$$

It relates a model to itself: the truth of p in a model does not affect the meaning of the specification.

15.2.4 Predicate

The set of models defining the meaning of a predicate is determined from the values of its constituent expressions. The set therefore depends on the particular treatment of undefinedness.

15.2.4.1 Membership predicate

The membership predicate $e_1 \in e_2$ is *true* if and only if the value of e_1 is in the set that is the value of e_2.

$$[\![e_1 \in e_2]\!]^{\mathcal{P}} \quad = \quad \{M : Model \mid [\![e_1]\!]^{\mathcal{E}} M \in [\![e_2]\!]^{\mathcal{E}} M \bullet M\}$$

In terms of the semantic universe, it is *true* in those models in which the semantic value of e_1 is in the semantic value of e_2, and is *false* otherwise.

15.2.4.2 Truth predicate

A truth predicate is always *true*.

$$[\![\text{true}]\!]^{\mathcal{P}} \quad = \quad Model$$

In terms of the semantic universe, it is *true* in all models.

15.2.4.3 Negation predicate

The negation predicate $\neg\ p$ is *true* if and only if p is *false*.

$$[\![\neg\ p]\!]^{\mathcal{P}} \quad = \quad Model \setminus [\![p]\!]^{\mathcal{P}}$$

In terms of the semantic universe, it is *true* in all models except those in which p is *true*.

15.2.4.4 Conjunction predicate

The conjunction predicate $p_1 \wedge p_2$ is *true* if and only if p_1 and p_2 are *true*.

$$[\![p_1 \wedge p_2]\!]^{\mathcal{P}} \quad = \quad [\![p_1]\!]^{\mathcal{P}} \cap [\![p_2]\!]^{\mathcal{P}}$$

In terms of the semantic universe, it is *true* in those models in which both p_1 and p_2 are *true*, and is *false* otherwise.

15.2.4.5 Universal quantification predicate

The universal quantification predicate $\forall\ e \bullet p$ is *true* if and only if predicate p is *true* for all bindings of the schema e.

$$[\![\forall\ e \bullet p]\!]^{\mathcal{P}} \quad = \quad \{M : Model \mid \forall\ t : [\![e]\!]^{\mathcal{E}} M \bullet M \oplus t \in [\![p]\!]^{\mathcal{P}} \bullet M\}$$

In terms of the semantic universe, it is *true* in those models for which p is *true* in that model overridden by all bindings in the semantic value of e, and is *false* otherwise.

15.2.5 Expression

Every expression has a semantic value, specified by the following semantic relations. The value of an undefined definite description expression is left loose, and hence the values of larger expressions containing undefined values are also loosely specified.

15.2.5.1 Reference expression

The value of the reference expression that refers to a non-generic definition i is the value of the declaration to which it refers.

$$[\![i]\!]^{\varepsilon} = \lambda M : Model \bullet M\ i$$

In terms of the semantic universe, its semantic value, given a model M, is that associated with the name i in M.

15.2.5.2 Generic instantiation expression

The value of the generic instantiation expression $i\ [e_1, ..., e_n]$ is a particular instance of the generic referred to by name i.

$$[\![i\ [e_1, ..., e_n]]\!]^{\varepsilon} = \lambda M : Model \bullet M\ i\ ([\![e_1]\!]^{\varepsilon} M, \ ..., \ [\![e_n]\!]^{\varepsilon} M)$$

In terms of the semantic universe, its semantic value, given a model M, is the generic value associated with the name i in M instantiated with the semantic values of the instantiation expressions in M.

15.2.5.3 Set extension expression

The value of the set extension expression $\{ e_1, ..., e_n \}$ is the set containing the values of its expressions.

$$[\![\{ e_1, ..., e_n \}]\!]^{\varepsilon} = \lambda M : Model \bullet \{[\![e_1]\!]^{\varepsilon} M, \ ..., [\![e_n]\!]^{\varepsilon} M\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set whose members are the semantic values of the member expressions in M.

15.2.5.4 Set comprehension expression

The value of the set comprehension expression $\{ e_1 \bullet e_2 \}$ is the set of values of e_2 for all bindings of the schema e_1.

$$[\![\{ e_1 \bullet e_2 \}]\!]^{\varepsilon} = \lambda M : Model \bullet \{t_1 : [\![e_1]\!]^{\varepsilon} M \bullet [\![e_2]\!]^{\varepsilon} (M \oplus t_1)\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of values of e_2 in M overridden with a binding value of e_1 in M.

15.2.5.5 Powerset expression

The value of the powerset expression $\mathbb{P}\, e$ is the set of all subsets of the set that is the value of e.

$$[\![\mathbb{P}\, e]\!]^{\varepsilon} = \lambda M : Model \bullet \mathbb{P}\, ([\![e]\!]^{\varepsilon} M)$$

In terms of the semantic universe, its semantic value, given a model M, is the powerset of values of e in M.

15.2.5.6 Tuple extension expression

The value of the tuple extension expression $(e_1, ..., e_n)$ is the tuple containing the values of its expressions in order.

$$[\![(e_1, ..., e_n)]\!]^{\varepsilon} = \lambda M : Model \bullet ([\![e_1]\!]^{\varepsilon} M, \ ..., \ [\![e_n]\!]^{\varepsilon} M)$$

In terms of the semantic universe, its semantic value, given a model M, is the tuple whose components are the semantic values of the component expressions in M.

15.2.5.7 Binding extension expression

The value of the binding extension expression $\langle\!|\ i_1 == e_1, ..., i_n == e_n\ |\!\rangle$ is the binding whose names are as enumerated and whose values are those of the associated expressions.

$$[\![\langle\!|\ i_1 == e_1, ..., i_n == e_n\ |\!\rangle]\!]^{\varepsilon} = \lambda M : Model \bullet \{i_1 \mapsto [\![e_1]\!]^{\varepsilon} M, \ ..., \ i_n \mapsto [\![e_n]\!]^{\varepsilon} M\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of pairs enumerated by its names each associated with the semantic value of the associated expression in M.

15.2.5.8 Definite description expression

The value of the definite description expression $\mu\ e_1 \bullet e_2$ is the sole value of e_2 that arises whichever binding is chosen from the set that is the value of schema e_1.

$$\begin{array}{l} \{M : Model;\ t_1 : \mathbb{W} \\ \quad \mid t_1 \in [\![\ e_1\ ]\!]^{\mathcal{E}} M \\ \quad \wedge\ (\forall\ t_3 : [\![\ e_1\ ]\!]^{\mathcal{E}} M \bullet [\![\ e_2\ ]\!]^{\mathcal{E}}\ (M \oplus t_3) = [\![\ e_2\ ]\!]^{\mathcal{E}}\ (M \oplus t_1)) \\ \bullet\ M \mapsto [\![\ e_2\ ]\!]^{\mathcal{E}}\ (M \oplus t_1)\} \end{array} \quad \subseteq\ [\![\ \mu\ e_1 \bullet e_2\ ]\!]^{\mathcal{E}}$$

In terms of the semantic universe, its semantic value, given a model M in which the value of e_2 in that model overridden by a binding of the schema e_1 is the same regardless of which binding is chosen, is that value of e_2. In other models, it has a semantic value, but this loose definition of the semantics does not say what it is.

15.2.5.9 Variable construction expression

The value of the variable construction expression $[i : e]$ is the set of all bindings whose sole name is i and whose associated value is in the set that is the value of e.

$$[\![\ [i : e]\ ]\!]^{\mathcal{E}} \ =\ \lambda\ M : Model \bullet \{w : [\![\ e\ ]\!]^{\mathcal{E}} M \bullet \{i \mapsto w\}\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of all singleton bindings (sets of pairs) of the name i associated with a value from the set that is the semantic value of e in M.

15.2.5.10 Schema construction expression

The value of the schema construction expression $[e \mid p]$ is the set of all bindings of schema e that satisfy the constraints of predicate p.

$$[\![\ [e \mid p]\ ]\!]^{\mathcal{E}} \ =\ \lambda\ M : Model \bullet \{t : [\![\ e\ ]\!]^{\mathcal{E}} M \mid M \oplus t \in [\![\ p\ ]\!]^{\mathcal{P}} \bullet t\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the bindings (sets of pairs) that are members of the semantic value of schema e in M such that p is *true* in the model M overridden with that binding.

15.2.5.11 Schema negation expression

The value of the schema negation expression $\neg\ e$ is that set of bindings that are of the same type as those in schema e but that are not in schema e.

$$[\![\ \neg\ e\ \mathbin{⦂}\ \mathbb{P}\tau\ ]\!]^{\mathcal{E}} \ =\ \lambda\ M : Model \bullet \{t : [\![\ \tau\ ]\!]^{\mathcal{T}} M \mid \neg\ t \in [\![\ e\ ]\!]^{\mathcal{E}} M \bullet t\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the bindings (sets of pairs) that are members of the semantic value of the carrier set of schema e in M such that those bindings are not members of the semantic value of schema e in M.

15.2.5.12 Schema conjunction expression

The value of the schema conjunction expression $e_1 \wedge e_2$ is the schema resulting from merging the signatures of schemas e_1 and e_2 and conjoining their constraints.

$$[\![\ e_1 \wedge e_2\ \mathbin{⦂}\ \mathbb{P}\tau\ ]\!]^{\mathcal{E}} \ =\ \lambda\ M : Model \bullet \{t : [\![\ \tau\ ]\!]^{\mathcal{T}} M;\ t_1 : [\![\ e_1\ ]\!]^{\mathcal{E}} M;\ t_2 : [\![\ e_2\ ]\!]^{\mathcal{E}} M \mid t_1 \cup t_2 = t \bullet t\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the unions of the bindings (sets of pairs) in the semantic values of e_1 and e_2 in M.

15.2.5.13 Schema universal quantification expression

The value of the schema universal quantification expression $\forall\ e_1 \bullet e_2$ is the set of bindings of schema e_2 restricted to exclude names that are in the signature of e_1, for all bindings of the schema e_1.

$$[\![\ \forall\ e_1 \bullet e_2\ \mathbin{⦂}\ \mathbb{P}\tau\ ]\!]^{\mathcal{E}} \ =\ \lambda\ M : Model \bullet \{t_2 : [\![\ \tau\ ]\!]^{\mathcal{T}} M \mid \forall\ t_1 : [\![\ e_1\ ]\!]^{\mathcal{E}} M \bullet t_1 \cup t_2 \in [\![\ e_2\ ]\!]^{\mathcal{E}}\ (M \oplus t_1) \bullet t_2\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the bindings (sets of pairs) in the semantic values of the carrier set of the type of the entire schema universal quantification expression in M, for which the union of the bindings (sets of pairs) in e_1 and in the whole expression is in the set that is the semantic value of e_2 in the model M overridden with the binding in e_1.

15.2.5.14 Schema renaming expression

The value of the schema renaming expression $e\ [j_1 \mathbin{/} i_1, ..., j_n \mathbin{/} i_n]$ is that schema whose bindings are like those of schema e except that some of its names have been replaced by new names, possibly merging components.

$$\begin{aligned}[\![\ e\ [j_1 \mathbin{/} i_1, ..., j_n \mathbin{/} i_n]\ ]\!]^{\mathcal{E}} \quad = \quad & \lambda\ M : Model \bullet \\ & \{t_1 : [\![\ e\ ]\!]^{\mathcal{E}} M;\ t_2 : \mathbb{W} \mid \\ & \quad t_2 = \{j_1 \mapsto i_1, ..., j_n \mapsto i_n\} \fatsemi t_1 \cup \{i_1, ..., i_n\} \lhd t_1 \\ & \quad \land\ t_2 \in (\_ \pfun \_) \\ & \bullet\ t_2\}\end{aligned}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the bindings (sets of pairs) in the semantic value of e in M with the new names replacing corresponding old names. Where components are merged by the renaming, those components shall have the same value.

15.2.6 Type

The value of a type is its carrier set.

NOTE 1 For an expression e with a defined value, $[\![\ e \fatcolon \tau\ ]\!]^{\mathcal{E}} \in [\![\ \tau\ ]\!]^{\mathcal{T}}$.

NOTE 2 The value of a generic type, $[\![\ [i_1, ..., i_n]\ \tau\ ]\!]^{\mathcal{T}}$, is never needed, and so is not defined.

NOTE 3 $[\![\ \tau\ ]\!]^{\mathcal{T}} M$ differs from *carrier* τ in that the former application returns a semantic value whereas the latter application returns an annotated parse tree.

15.2.6.1 Given type

$$[\![\ \texttt{GIVEN}\ i\ ]\!]^{\mathcal{T}} \quad = \quad \lambda\ M : Model \bullet M\ (i\ decor\ \heartsuit)$$

The semantic value of the given type `GIVEN` i, given a model M, is the semantic value associated with the given type name i in M.

15.2.6.2 Generic parameter type

$$[\![\ \texttt{GENTYPE}\ i\ ]\!]^{\mathcal{T}} \quad = \quad \lambda\ M : Model \bullet M\ (i\ decor\ \spadesuit)$$

The semantic value of the generic type `GENTYPE` i, given a model M, is the semantic value associated with generic parameter name i in M.

15.2.6.3 Powerset type

$$[\![\ \mathbb{P}\,\tau\ ]\!]^{\mathcal{T}} \quad = \quad \lambda\ M : Model \bullet \mathbb{P}\ ([\![\ \tau\ ]\!]^{\mathcal{T}} M)$$

The semantic value of the set type $\mathbb{P}\,\tau$, given a model M, is the powerset of the semantic value of type τ in M.

15.2.6.4 Cartesian product type

$$[\![\ \tau_1 \times ... \times \tau_n\ ]\!]^{\mathcal{T}} \quad = \quad \lambda\ M : Model \bullet ([\![\ \tau_1\ ]\!]^{\mathcal{T}} M) \times ... \times ([\![\ \tau_n\ ]\!]^{\mathcal{T}} M)$$

The semantic value of the Cartesian product type $\tau_1 \times ... \times \tau_n$, given a model M, is the Cartesian product of the semantic values of types $\tau_1 ... \tau_n$ in M.

15.2.6.5 Schema type

$$[\![\ [i_1 : \tau_1;\ ...;\ i_n : \tau_n]\ ]\!]^{\mathcal{T}} \ = \ \lambda\ M : Model \bullet \{t : \{i_1,\ ...,\ i_n\} \to \mathbb{W} \mid t\ i_1 \in [\![\ \tau_1\ ]\!]^{\mathcal{T}} M \wedge ... \wedge t\ i_n \in [\![\ \tau_n\ ]\!]^{\mathcal{T}} M \bullet t\}$$

The semantic value of the schema type $[i_1 : \tau_1;\ ...;\ i_n : \tau_n]$, given a model M, is the set of bindings, represented by sets of pairs of names and values, for which the names are those of the schema type and the associated values are the semantic values of the corresponding types in M.

Annex A
(normative)

Mark-ups

A.1 Introduction

Not all systems support ISO/IEC 10646 [5][6] (UCS), the definitive representation of Z characters (clause 6). Other representations are known as mark-ups. For each mark-up, there shall be a functional mapping from it to the UCS representation. The UCS representation may be used directly: the identity function is a conformant mapping. This annex defines two mark-ups based on 7-bit ASCII [4] and their mappings:

- a LaTeX [10] mark-up, suitable for processing by that tool to render Z characters in their mathematical form;
- an e-mail, or lightweight ASCII, mark-up, suitable for rendering Z characters on a low resolution device, such as an ASCII-character-based terminal, or in e-mail correspondence.

The mark-up mappings described in this annex show how to partition ASCII mark-up into 'mark-up tokens', and how to convert each mark-up token to a corresponding sequence of Z characters. Mark-up tokens that are individual ASCII characters (such as digits) are converted by default directly to the corresponding Z character, i.e. from ASCII-*xy* to `0000 00`*xy* in UCS. Use of different sequences of mark-up characters that correspond to the same Z characters is permitted by this International Standard, as the same lexical tokens will result. However, tools may require lexical tokens to be marked-up consistently.

Some mark-ups, such as the LaTeX mark-up, have a rendering separate from their conversion. A mark-up may specify rendering information, such as bold or italic, that is irrelevant to its conversion. A mark-up's conversion and rendering shall be consistent with each other, i.e. someone reading the rendering should perceive the same sequence of lexical tokens as is specified by this International Standard from the conversion.

The semantics of a specification are specified by the normative clauses of this International Standard on the assumption that its sections are in a definition before use order. This restriction need not be imposed on the order in which sections are presented to a human reader or to a tool. A mark-up shall be such that section headers can be recognised without recognising anything else, so that sections marked-up out-of-order can be permuted first.

NOTE 1 Specifications with cycles in the parents relation are erroneous, so there is no need to permute their sections.

NOTE 2 The syntax of a mark-up is separate from the syntax of the lexis (clause 7). It would be a mistake to assume that a mark-up should follow the syntax of the lexis.

EXAMPLE The LaTeX mark-up `\begin{schema}{S}` becomes the Z characters `SCHCHAR S` for recognition by the lexis, the braces not appearing in the Z characters, though they have to be there in the LaTeX mark-up.

A.2 LaTeX mark-up

NOTE Some guidance on implementing the specification contained in this clause is given in [17].

A.2.1 Rendering mode

The rendering of mark-up by LaTeX depends on the mode in which it is interpreted. The mark-up of formal Z text shall be interpreted and rendered in LaTeX's math mode.

Other aspects of the rendering shall be defined to be consistent with the conversion defined below.

A.2.2 White space and comments

Within LaTeX's math mode, white space separates tokens of the mark-up, but only sometimes is rendered as space.

The ASCII white space characters space, tab and newline are 'soft': they render as nothing, and they shall be converted to no Z characters.

The 'hard' space mark-ups render as specific quantities of space and shall be converted to Z characters as listed below. The LaTeX commands for hard space shall be in scope throughout a specification.

NOTE 1 Early ones in the table are existing LaTeX commands; later ones (the tab stops onwards) are additional commands that have traditionally been used in mark-up of Z.

LaTeX command	Rendering	Z character
~	interword space	SPACE
\,	thin space	SPACE
\:	medium space	SPACE
\;	thick space	SPACE
\(space)	interword space	SPACE
\\	newline	NLCHAR
\t1	tab stop 1	SPACE
\t2	tab stop 2	SPACE
\t3	tab stop 3	SPACE
\t4	tab stop 4	SPACE
\t5	tab stop 5	SPACE
\t6	tab stop 6	SPACE
\t7	tab stop 7	SPACE
\t8	tab stop 8	SPACE
\t9	tab stop 9	SPACE
\also	small vertical space	NLCHAR
\znewpage	new page	NLCHAR

NOTE 2 It is the generated SPACE characters that go on to separate the tokens of the Z lexis.

All LaTeX comments, except directives (see A.2.3), shall be ignored.

NOTE 3 A LaTeX comment starts with a % character and extends to the end of that line, over the newline character itself, and ends with any spaces and tabs at the start of the next line.

All { and } characters that are not prefixed with \ can affect the conversion of neighbouring mark-up, but shall themselves convert to no Z characters.

A.2.3 Mark-up directives

A.2.3.1 Introduction

LaTeX mark-up is based on commands whose definitions determine their rendering. The conversion of the same commands to corresponding sequences of Z characters shall be specified by mark-up directives (except the commands for hard-space defined in A.2.2 above).

Rendering definitions are in terms of either a specific character or a sequence of other existing mark-up. Mark-up directives shall analogously be in terms of either a specific character or a sequence of other existing mark-up.

NOTE Rendering definitions are typically formulated in terms of \DeclareMathSymbol and \newcommand commands.

Commands that are used as operator words have space rendered between them and their operands. Mark-up directives shall specify the conversion of corresponding space. Whereas the quantity of space rendered depends on whether the operator is a function or a relation, a single SPACE character in the conversion suffices to ensure consistency between rendering and conversion.

Just as it is an error for a command to be used without a rendering definition, it shall be an error for a command to be used without a mark-up directive.

In general, a directive begins with %% at the start of a line (no leading space), and ends at the end of that line.

The usual LaTeX rule concerning soft space following a LaTeX command being ignored is disabled in the case of a newline character at the end of a directive.

A.2.3.2 Mark-up directives for single characters

The `%%Zchar` mark-up directive specifies the conversion of a LaTeX command to a Z character. It takes one of the following forms,

```
%%Zchar \LaTeXcommand U+nnnn
%%Zchar \LaTeXcommand U-nnnnnnnn
```

where `nnnn` is four hexadecimal digits identifying the code position of a character in the Basic Multilingual Plane of UCS (`0000 nnnn`), and `nnnnnnnn` is eight hexadecimal digits identifying the code position of a character anywhere in UCS (`nnnn nnnn`). A use of this `\LaTeXcommand` shall be converted to the corresponding character.

EXAMPLE 1
```
%%Zchar \Delta U+0394
%%Zchar \arithmos U-0001D538
```

The LaTeX mark-up '`\Delta`' converts to the Z character 'Δ' and renders as 'Δ'.
The LaTeX mark-up '`\arithmos`' converts to the Z character '$\mathbb{A}$' and renders as '$\mathbb{A}$'.

If a `\LaTeXcommand` for a character is used as an operator word, then the conversion of `SPACE` characters before and/or after it shall be specified using the appropriate `%%Zprechar`, `%%Zpostchar` or `%%Zinchar` mark-up directive. Any following subscript or superscript shall precede any following `SPACE`. These `SPACE` characters shall be omitted if the `\LaTeXcommand` is enclosed in braces.

EXAMPLE 2
```
%%Zinchar \sqsubseteq U+2291
%%Zprechar \finset U-0001D53D
```

The LaTeX mark-up '`\sqsubseteq`' converts to the Z character sequence '$\sqsubseteq$ ' and renders as '$\sqsubseteq$'.
The LaTeX mark-up '`\sqsubseteq_1`' converts to the Z character sequence '$\sqsubseteq \searrow 1 \nwarrow$ ' and renders as '$\sqsubseteq_1$'.
The LaTeX mark-up '`\sqsubseteq~_1`' converts to the Z character sequence '$\sqsubseteq \searrow 1 \nwarrow$' and renders as '$\sqsubseteq\ _1$'.
The LaTeX mark-up '`\finset`' converts to the Z character sequence '$\mathbb{F}$ ' and renders as '$\mathbb{F}$'.
The LaTeX mark-up '`{\finset}`' converts to the Z character sequence '$\mathbb{F}$' and renders as '$\mathbb{F}$'.

NOTE The braced form is necessary when the character is used within a larger word.

A.2.3.3 Mark-up directives for words

The `%%Zword` mark-up directive specifies the conversion of a LaTeX command to a sequence of Z characters. It takes the following form,

```
%%Zword \LaTeXcommand Zstring
```

where `Zstring` is LaTeX mark-up for a sequence of Z characters. The `Zstring` shall exclude any leading soft spaces and end at the end of the line. A use of this `\LaTeXcommand` shall be converted to the conversion of `Zstring`.

EXAMPLE 1
```
%%Zword \phiS \phi S
```

The LaTeX mark-up '`\phiS`' converts to the Z character sequence 'ϕS' and renders as 'ϕS'.

If a `\LaTeXcommand` for a word is used as an operator word, then the conversion of `SPACE` characters before and/or after it shall be specified using the appropriate `%%Zpreword`, `%%Zpostword` or `%%Zinword` mark-up directive. Any

following subscript or superscript shall precede any following SPACE. These SPACE characters shall be omitted if the \LaTeXcommand is enclosed in braces.

EXAMPLE 2
```
%%Zinword \dcat {\cat}/
```

The LaTeX mark-up '\dcat' converts to the Z character sequence ' ⌢/ ' and renders as '⌢/'.
The LaTeX mark-up '\dcat_1' converts to the Z character sequence ' ⌢/↘1↖ ' and renders as '⌢/$_1$'.
The LaTeX mark-up '\dcat~_1' converts to the Z character sequence ' ⌢/ ↘1↖' and renders as '⌢/ $_1$'.
The LaTeX mark-up '{\dcat}' converts to the Z character sequence '⌢/'and renders as '⌢/'.

NOTE The braced form is necessary when the character sequence is used within a larger word.

A.2.3.4 Scope of a mark-up directive

The scope of a mark-up directive is the entire section in which it appears and any sections of which it is an ancestor, excluding the headers of those sections, and excepting earlier such directives. There can be no more than one directive for a particular \LaTeXcommand in the same scope. These %% directives may appear within as well as between formal paragraphs, merely having to be on lines by themselves.

In converting the Zstring of the mark-up directive for a word, the mark-up rules defined for the Z core language, and the mark-up rules established by earlier mark-up directives, shall be applied.

EXAMPLE
```
%%Zword \foo x \bar y
%%Zword \bar +
```

The order in which these two directives are written matters: in this order, either \bar is erroneously used before it is defined, or \bar is erroneously defined more than once.

A.2.4 Core characters and words

The following mark-up shall be provided for the characters and words of the Z core language. Each LaTeX command and its conversion is specified in the form of a directive that shall appear in the mark-up of the prelude section.

A.2.4.1 Greek alphabet characters

Only the minimal subset of Greek alphabet defined in 6.2 needs to be supported by an implementation. The mark-ups of any other Greek characters shall be as if specified by %%Zchar directives.

Z character	LaTeX mark-up
Δ	%%Zchar \Delta U+0394
Ξ	%%Zchar \Xi U+039E
θ	%%Zprechar \theta U+03B8
λ	%%Zprechar \lambda U+03BB
μ	%%Zprechar \mu U+03BC

EXAMPLE
The LaTeX mark-up '\Delta S' converts to the Z character sequence 'ΔS' and renders as 'ΔS'.
The LaTeX mark-up '\Delta~S' converts to the Z character sequence 'Δ S' and renders as '$\Delta\ S$'.
The LaTeX mark-up '\lambda x' converts to the Z character sequence 'λ x' and renders as '$\lambda\, x$'.
The LaTeX mark-up '{\lambda}x' converts to the Z character sequence 'λx' and renders as 'λx'.

NOTE 1 Exceptions are made for θ, λ and μ to ease the mark-up of binding construction, function construction and definite description expressions.

NOTE 2 LaTeX does not provide separate mark-up for upper case Greek letters that render like Roman counterparts.

A.2.4.2 Other letter characters

Z character	LaTeX mark-up
$\mathbb{A}$	`%%Zchar \arithmos U-0001D538`
$\mathbb{N}$	`%%Zchar \nat U+2115`
$\mathbb{P}$	`%%Zprechar \power U+2119`

EXAMPLE
The LaTeX mark-up '`\power S`' converts to the Z character sequence '$\mathbb{P}\ S$' and renders as '$\mathbb{P}\, S$'.
The LaTeX mark-up '`{\power}S`' converts to the Z character sequence '$\mathbb{P}S$' and renders as '$\mathbb{P}S$'.

A.2.4.3 Special characters

Z character	LaTeX mark-up
_	`%%Zchar \_ U+005F`
{	`%%Zchar \{ U+007B`
}	`%%Zchar \} U+007D`
⦇	`%%Zchar \ldata U+300A`
⦈	`%%Zchar \rdata U+300B`
⦉	`%%Zchar \lblot U+2989`
⦊	`%%Zchar \rblot U+298A`

NOTE No `SPACE` characters need be converted around `SPECIAL` characters; space may be rendered if desired.

The mark-up `\_` needs to be in scope in section headers, so is not introduced in the prelude. It is in scope throughout a specification.

The LaTeX mark-up for subscripts and superscripts shall be converted as follows.

LaTeX mark-up	Z characters
`^` ⟨single LaTeX token⟩	↗ ⟨Z character sequence⟩ ↙
`^{` ⟨LaTeX tokens⟩ `}`	↗ ⟨Z character sequence⟩ ↙
`_` ⟨single LaTeX token⟩	↘ ⟨Z character sequence⟩ ↖
`_{` ⟨LaTeX tokens⟩ `}`	↘ ⟨Z character sequence⟩ ↖

EXAMPLE
LaTeX mark-up `x^1` converts to Z character sequence '`x`↗`1`↙', and renders as 'x^1'
LaTeX mark-up `x^{1}` converts to Z character sequence '`x`↗`1`↙', and renders as 'x^1'
LaTeX mark-up `x^\Delta` converts to Z character sequence '`x`↗Δ↙', and renders as 'x^{Δ}'
LaTeX mark-up `x^{\Delta S}` converts to Z character sequence '`x`↗Δ`S`↙', and renders as '$x^{\Delta S}$'
LaTeX mark-up `\exists_1` converts to Z character sequence '∃↘`1`↖', and renders as '$\exists_1$'
LaTeX mark-up `\exists_{1}` converts to Z character sequence '∃↘`1`↖', and renders as '$\exists_1$'
LaTeX mark-up `\exists_\Delta` converts to Z character sequence '∃↘Δ↖', and renders as '$\exists_{\Delta}$'
LaTeX mark-up `\exists_{\Delta S}` converts to Z character sequence '∃↘Δ`S`↖', and renders as '$\exists_{\Delta S}$'
LaTeX mark-up `x_a^b` converts to Z character sequence '`x`↘`a`↖↗`b`↙', and renders as '$x_a{}^b$'
LaTeX mark-up `x_{a^b}` converts to Z character sequence '`x`↘`a`↗`b`↙↖', and renders as 'x_{a^b}'
LaTeX mark-up `x_a{}_b` converts to Z character sequence '`x`↘`a`↖↘`b`↖', and renders as '$x_a{}_b$'

Box characters arise from the conversion of paragraph mark-up, as described in A.2.7. Mark-ups for `NLCHAR` and `SPACE` are given in A.2.2.

A.2.4.4 Symbol characters (except mathematical toolkit characters)

The LaTeX mark-up of the Z character • is the ASCII character `@` (see below for its spacing behaviour.) Its mark-up is in scope throughout a specification.

Z character	LaTeX mark-up
⊢	`%%Zchar \vdash U+22A2`
∧	`%%Zinchar \land U+2227`
∨	`%%Zinchar \lor U+2228`
⇒	`%%Zinchar \implies U+21D2`
⇔	`%%Zinchar \iff U+21D4`
¬	`%%Zprechar \lnot U+00AC`
∀	`%%Zprechar \forall U+2200`
∃	`%%Zprechar \exists U+2203`
×	`%%Zinchar \cross U+00D7`
∈	`%%Zinchar \in U+2208`
\	`%%Zinchar \hide U+29F9`
↾	`%%Zinchar \project U+2A21`
⨾	`%%Zinchar \semi U+2A1F`
≫	`%%Zinchar \pipe U+2A20`

Some of the ASCII symbol characters have an associated spacing class in LaTeX's math mode, which causes space to be rendered around them in certain contexts.

The characters `* + - @` and `|` are classed as LaTeX math functions. `SPACE` characters shall usually be added around their individual occurrences. Any following subscript or superscript shall precede the following `SPACE`. The cases where the surrounding `SPACE`s are omitted are when the character is enclosed in braces and when the character is itself used as a subscript or superscript (follows `_` or `^`).

EXAMPLE 1
The LaTeX mark-up '`x+y`' converts to the Z character sequence '`x + y`' and renders as '$x + y$'.
The LaTeX mark-up '`x{+}y`' converts to the Z character sequence '`x+y`' and renders as '$x+y$'.
The LaTeX mark-up '`x +_1 y`' converts to the Z character sequence '`x +`↘`1`↖ `y`' and renders as '$x +_1 y$'.
The LaTeX mark-up '`x^+`' converts to the Z character sequence '`x`↗`+`↙' and renders as 'x^+'.

The characters `;` and `,` are classed as LaTeX math punctuations. They are converted like the LaTeX math functions above, except that no `SPACE` is added before their occurrences.

The characters `:` `<` `=` and `>` are classed as LaTeX math relations. `SPACE` characters shall usually be added around sequences of their occurrences. Any following subscript or superscript shall precede the following `SPACE`. The cases where the surrounding `SPACE`s are omitted are when the sequence of characters is enclosed in braces and when the sequence of characters is itself used as a subscript or superscript (follows `_` or `^`).

EXAMPLE 2
The LaTeX mark-up '`x=y`' converts to the Z character sequence '`x = y`' and renders as '$x = y$'.
The LaTeX mark-up '`x==y`' converts to the Z character sequence '`x == y`' and renders as '$x == y$'.
The LaTeX mark-up '`x::=y`' converts to the Z character sequence '`x ::= y`' and renders as '$x ::= y$'.
The LaTeX mark-up '`x{=}y`' converts to the Z character sequence '`x=y`' and renders as '$x=y$'.
The LaTeX mark-up '`x =_1 y`' converts to the Z character sequence '`x =`↘`1`↖ `y`' and renders as '$x =_1 y$'.
The LaTeX mark-up '`x^=`' converts to the Z character sequence '`x`↗`=`↙' and renders as '$x^=$'.
The LaTeX mark-up '`x= =y`' converts to the Z character sequence '`x == y`' and renders as '$x == y$'.

A.2.4.5 Core words

Z characters	LaTeX mark-up
if	`%%Zpreword \IF if`
then	`%%Zinword \THEN then`
else	`%%Zinword \ELSE else`
let	`%%Zpreword \LET let`
pre	`%%Zpreword \pre pre`
function	`%%Zpreword \function function`
generic	`%%Zpreword \generic generic`
relation	`%%Zpreword \relation relation`
leftassoc	`%%Zinword \leftassoc leftassoc`
rightassoc	`%%Zinword \leftassoc leftassoc`

EXAMPLE LaTeX mark-up: `\IF \disjoint a \THEN x = y \mod z \ELSE x = y \div z`
Example rendering: if *disjoint a* then $x = y \mod z$ else $x = y \operatorname{div} z$
Z characters: `if disjoint a then x = y mod z else x = y div z`

NOTE 1 Some command names are capitalised to avoid clashes with existing command names.

NOTE 2 Mark-up directives for the section and parents keywords do not appear in the prelude section, as they would not be in scope in section headers where those LaTeX commands are used (see A.2.6).

A LaTeX command shall be provided for the core keyword , , to ease the avoidance of converted `SPACE` between the commas (see A.2.4.4). A LaTeX command shall be provided for the core keyword _ to ease the conversion of `SPACE`s around it.

Z characters	LaTeX mark-up
, ,	`%%Zinword \listarg {,}{,}`
_	`%%Zinword \varg \_`

A.2.5 Mathematical toolkit characters and words

The following mark-up shall be provided for the names defined in the mathematical toolkit. Each LaTeX command and its conversion is specified in the form of a directive that shall appear in the mark-up of the relevant section.

A.2.5.1 Section set_toolkit

Z character	LaTeX mark-up
$\leftrightarrow$	%%Zinchar \rel U+2194
$\rightarrow$	%%Zinchar \fun U+2192
$\neq$	%%Zinchar \neq U+2260
$\notin$	%%Zinchar \notin U+2209
$\varnothing$	%%Zchar \emptyset U+2205
$\subseteq$	%%Zinchar \subseteq U+2286
$\subset$	%%Zinchar \subset U+2282
$\cup$	%%Zinchar \cup U+222A
$\cap$	%%Zinchar \cap U+2229
$\setminus$	%%Zinchar \setminus U+005C
$\ominus$	%%Zinchar \symdiff U+2296
$\bigcup$	%%Zprechar \bigcup U+22C3
$\bigcap$	%%Zprechar \bigcap U+22C2
$\mathbb{F}$	%%Zprechar \finset U-0001D53D

A.2.5.2 Section relation_toolkit

Z characters	LaTeX mark-up
$\mapsto$	%%Zinchar \mapsto U+21A6
dom	%%Zpreword \dom dom
ran	%%Zpreword \ran ran
id	%%Zpreword \id id
$\fatsemi$	%%Zinchar \comp U+2A3E
$\circ$	%%Zinchar \circ U+2218
$\lhd$	%%Zinchar \dres U+25C1
$\rhd$	%%Zinchar \rres U+25B7
$\ntriangleleft$	%%Zinchar \ndres U+2A64
$\ntriangleright$	%%Zinchar \nrres U+2A65
$\sim$	%%Zpostchar \inv U+223C
$\limg$	%%Zinchar \limg U+2987
$\rimg$	%%Zpostchar \rimg U+2988
$\oplus$	%%Zinchar \oplus U+2295
$\nearrow+\swarrow$	%%Zpostword \plus ^+
$\nearrow*\swarrow$	%%Zpostword \star ^*

A.2.5.3 Section function_toolkit

Z characters	LaTeX mark-up
$\pfun$	%%Zinchar \pfun U+21F8
$\pinj$	%%Zinchar \pinj U+2914
$\inj$	%%Zinchar \inj U+21A3
$\psurj$	%%Zinchar \psurj U+2900
$\surj$	%%Zinchar \surj U+21A0
$\bij$	%%Zinchar \bij U+2916
$\ffun$	%%Zinchar \ffun U+21FB
$\finj$	%%Zinchar \finj U+2915
disjoint	%%Zpreword \disjoint disjoint
partition	%%Zinword \partition partition

A.2.5.4 Section number_toolkit

Z characters	LaTeX mark-up
$\mathbb{Z}$	%%Zchar \num U+2124
-	%%Zprechar \negate U+002D
$\leq$	%%Zinchar \leq U+2264
$\geq$	%%Zinchar \geq U+2265
div	%%Zinword \div div
mod	%%Zinword \mod mod

The mark-up character '-', having had SPACEs put around it because it is a LaTeX math function character (see A.2.4.4), shall be converted to the subtraction operator's Z character 'U+2212', while the Z character '-' shall be the name of the negation operator.

A.2.5.5 Section sequence_toolkit

Z characters	LaTeX mark-up
..	%%Zinword \upto ..
#	%%Zprechar \# U+0023
seq	%%Zpreword \seq seq
iseq	%%Zpreword \iseq iseq
$\langle$	%%Zprechar \langle U+3008
$\rangle$	%%Zpostchar \rangle U+3009
$\frown$	%%Zinchar \cat U+2040
$\upharpoonleft$	%%Zinchar \extract U+21BF
$\upharpoonright$	%%Zinchar \filter U+21BE
prefix	%%Zinword \prefix prefix
suffix	%%Zinword \suffix suffix
infix	%%Zinword \infix infix
$\frown/$	%%Zinword \dcat {\cat}/

The superscripted form of relational iteration shall be marked-up using `^{` and `}`, with hard space where necessary to prevent these being mistaken for parts of a larger word.

A.2.6 Section header mark-up

Section headers shall be enclosed in a LaTeX `zsection` environment. The `\begin{zsection}` is converted to `ZEDCHAR`. The `\end{zsection}` is converted to `ENDCHAR`.

```
\begin{zsection}
\SECTION NAME \parents ...
\end{zsection}
```

Within a section header, the only LaTeX commands that are converted are `\SECTION`, `\parents`, `\_` and the hard space commands of A.2.2. These mark-ups for the section and parents keywords shall be as if specified by the following mark-up directives, and shall be in scope throughout a specification.

Z characters	**LaTeX mark-up**
section	`%%Zpreword \SECTION section`
parents	`%%Zinword \parents parents`

A.2.7 Paragraph mark-up

A.2.7.1 Introduction

Each formal Z paragraph appears between a pair of `\begin{xxx}` and `\end{xxx}` LaTeX commands. Text not appearing between such commands is informal accompanying text.

The `\begin{xxx}` command is converted to a box character, the `ZEDCHAR` character serving for those paragraphs that are rendered without a box. Any middle line in a boxed paragraph is marked-up using the `\where` LaTeX command, which is converted to the Z | character with `SPACE`s around it. The `\end{xxx}` command is converted to the Z `ENDCHAR` character.

A.2.7.2 Axiomatic description paragraph mark-up

```
\begin{axdef}
DeclPart
\where
Predicate
\end{axdef}
```

The mark-up `\begin{axdef}` is converted to an `AXCHAR` character. The mark-up `\where` is converted to a | character with `SPACE`s around it. The mark-up `\end{axdef}` is converted to an `ENDCHAR` character.

A.2.7.3 Schema definition paragraph mark-up

```
\begin{schema}{NAME}
DeclPart
\where
Predicate
\end{schema}
```

The mark-up `\begin{schema}{ }` is converted to a `SCHCHAR` character. The mark-up `\where` is converted to a | character with `SPACE`s around it. The mark-up `\end{schema}` is converted to an `ENDCHAR` character.

A.2.7.4 Generic axiomatic description paragraph mark-up

```
\begin{gendef}[Formals]
DeclPart
\where
Predicate
\end{gendef}
```


The mark-up `\begin{gendef}` is converted to an `AXCHAR` and a `GENCHAR` character. The mark-up `\where` is converted to a | character with `SPACE`s around it. The mark-up `\end{gendef}` is converted to an `ENDCHAR` character.

A.2.7.5 Generic schema definition paragraph mark-up

```
\begin{schema}{NAME}[Formals]
DeclPart
\where
Predicate
\end{schema}
```

The mark-up `\begin{schema}{ }` is converted to a `SCHCHAR` and a `GENCHAR` character. The mark-up `\where` is converted to a | character with `SPACE`s around it. The mark-up `\end{schema}` is converted to an `ENDCHAR` character.

A.3 E-mail mark-up

This e-mail mark-up is designed primarily as a human-readable lightweight mark-up, but may also be processed by tools. The character '`%`' delimits an ASCII string used to represent a sequence of Z characters, for example '×' as '`%x%`'. This disambiguates it from, for example, the name '`x`'.

Where there is no danger of ambiguity (for the human reader) the trailing '`%`' character, or both '`%`' characters, may be omitted to reduce clutter.

A literal '`%`' character may be introduced into the text as '`%%`'.

A.3.1 Letter characters

In the following, the e-mail string is to be used surrounded by a leading and trailing '`%`' character.

Names that use only ASCII characters, or that are composed out of previously defined Z characters, are not listed here.

A.3.1.1 Greek alphabet characters

Only the minimal subset of Greek alphabet defined in 6.2 need be supported by an implementation. Those Greek characters that are supported shall use the mark-up given here.

E-mail string	Z character
`Delta`	Δ
`Xi`	Ξ
`theta`	θ
`lambda`	λ
`mu`	μ

A.3.1.2 Other Z core language letter characters

E-mail string	Z character
`arithmos`	$\mathbb{A}$
`N`	$\mathbb{N}$
`P`	$\mathbb{P}$

A.3.2 Special characters

E-mail string	Z character
/^	↗
v/	↙
\v	↘
^\	↖
<<	⟪
>>	⟫
<|	⦇
|>	⦈

Box characters arise from the conversion of paragraph mark-up, as described in A.3.5.

A.3.3 Symbol characters (except mathematical toolkit characters)

E-mail string	Z character
|	|
|-	⊢
/\	∧
\/	∨
==>	⇒
<=>	⇔
not	¬
A	∀
E	∃
x	×
e	∈
@	•
S\	\
S|\	↾
S;	⨾
S>>	≫

A.3.4 Mathematical toolkit characters and words

The following mark-up is provided for the tokens in the mathematical toolkit.

A.3.4.1 Section set_toolkit

E-mail string	Z character
<-->	$\leftrightarrow$
-->	$\rightarrow$
/=	$\neq$
/e	$\notin$
(/)	$\varnothing$
c_	$\subseteq$
c	$\subset$
u	$\cup$
n	$\cap$
(-)	$\ominus$
uu	$\bigcup$
nn	$\bigcap$
F	$\mathbb{F}$

A.3.4.2 Section relation_toolkit

E-mail string	Z character
|-->	$\mapsto$
;	$\fatsemi$
o	$\circ$
<:	$\lhd$
:>	$\rhd$
<-:	$\unlhd$
:->	$\unrhd$
~	$\sim$
(|	$(\!\mid$
|)	$\mid\!)$
(+)	$\oplus$
+	$+$
*	$*$

A.3.4.3 Section function_toolkit

E-mail string	Z character
-|->	$\pfun$
>-|->	$\inj$
>-->	$\rightarrowtail$
-|->>	$\surj$
-->>	$\twoheadrightarrow$
>-->>	$\bij$
-||->	$\finfun$
>-||->	$\finfun1$

A.3.4.4 Section number_toolkit

E-mail string	Z character
Z	$\mathbb{Z}$
-	- (unary negation)
<=	$\leq$
>=	$\geq$

A.3.4.5 Section sequence_toolkit

E-mail string	Z character
<	$\langle$
>	$\rangle$
^	$\frown$
/|	$\upharpoonleft$
|\	$\upharpoonright$

A.3.5 Paragraph mark-up

A.3.5.1 Axiomatic description paragraph mark-up

```
+..
DeclPart
|--
Predicate
-..
```

The mark-up `+..` is converted to an `AXCHAR` character. The mark-up `|--` is converted to a | character. The mark-up `-..` is converted to an `ENDCHAR` character.

A.3.5.2 Schema definition paragraph mark-up

```
+-- NAME ---
DeclPart
|--
Predicate
---
```

The mark-up `+--` `---` is converted to a `SCHCHAR` character. The mark-up `|--` is converted to a | character. The mark-up `---` is converted to an `ENDCHAR` character.

A.3.5.3 Generic axiomatic description paragraph mark-up

```
+== [Formals] ===
DeclPart
|--
Predicate
-==
```

The mark-up `+==` `===` is converted to an `AXCHAR` and a `GENCHAR` character. The mark-up `|--` is converted to a | character. The mark-up `-==` is converted to an `ENDCHAR` character.

A.3.5.4 Generic schema definition paragraph mark-up

```
+-- NAME[Formals] ---
DeclPart
|--
Predicate
---
```

The mark-up `+--  ---` is converted to a `SCHCHAR` and a `GENCHAR` character. The mark-up `|--` is converted to a | character. The mark-up `---` is converted to an `ENDCHAR` character.

A.3.5.5 Other paragraph mark-up

Unboxed formal paragraphs start with '`%%Z`' on a line by itself, which converts to a `ZEDCHAR` character, and end with a '`%%`' on a line by itself, which converts to an `ENDCHAR` character.

Annex B
(normative)

Mathematical toolkit

B.1 Introduction

The mathematical toolkit is an optional extension to the compulsory Z core language. It comprises a hierarchy of related sections, each defining operators that are widely used in common application domains.

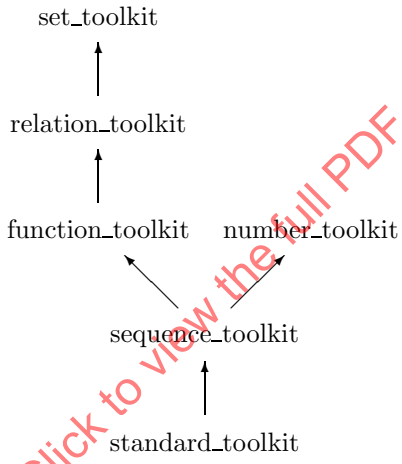


Figure B.1 — Parent relation between sections of the mathematical toolkit

The division of the mathematical toolkit into separate sections allows use of certain subsets of the toolkit rather than its entirety. For example, if sequences are not used in a particular specification, then using *function_toolkit* and *number_toolkit* as parents avoids bringing the notation of *sequence_toolkit* into scope. Notations that are not reused can be given different definitions.

Section *standard_toolkit* is an implicit parent of an anonymous specification.

B.2 Preliminary definitions

section *set_toolkit*

B.2.1 Relations

generic 5 rightassoc $(\_ \leftrightarrow \_)$

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

$X \leftrightarrow Y$ is the set of relations between X and Y, that is, the set of all sets of ordered pairs whose first members are members of X and whose second members are members of Y.

B.2.2 Total functions

generic 5 rightassoc $(\_ \to \_)$

$$X \to Y == \{ f : X \leftrightarrow Y \mid \forall x : X \bullet \exists_1 y : Y \bullet (x, y) \in f \}$$

$X \to Y$ is the set of all total functions from X to Y, that is, the set of all relations between X and Y such that each x in X is related to exactly one y in Y.

B.3 Sets

B.3.1 Inequality relation

relation $(\_ \neq \_)$

$$[X]$$
$$\_ \neq \_ : X \leftrightarrow X$$
$$\forall x, y : X \bullet x \neq y \Leftrightarrow \neg\, x = y$$

Inequality is the relation between those values of the same type that are not equal to each other.

B.3.2 Non-membership

relation $(\_ \notin \_)$

$$[X]$$
$$\_ \notin \_ : X \leftrightarrow \mathbb{P}\, X$$
$$\forall x : X;\ a : \mathbb{P}\, X \bullet x \notin a \Leftrightarrow \neg\, x \in a$$

Non-membership is the relation between those values of a type, x, and sets of values of that type, a, for which x is not a member of a.

B.3.3 Empty set

$$\varnothing[X] == \{ x : X \mid \mathit{false} \}$$

The empty set of any type is the set of that type that has no members.

B.3.4 Subset relation

relation $(\_ \subseteq \_)$

$$[X]$$
$$\_ \subseteq \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X$$
$$\forall a, b : \mathbb{P}\, X \bullet a \subseteq b \Leftrightarrow (\forall x : a \bullet x \in b)$$

Subset is the relation between two sets of the same type, a and b, such that every member of a is a member of b.

B.3.5 Proper subset relation

relation $(\_ \subset \_)$

$$[X]$$
$$\_ \subset \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X$$
$$\forall\, a, b : \mathbb{P}\, X \bullet a \subset b \Leftrightarrow a \subseteq b \land a \neq b$$

Proper subset is the relation between two sets of the same type, a and b, such that a is a subset of b and a and b are not equal.

B.3.6 Non-empty subsets

$$\mathbb{P}_1\, X == \{\ a : \mathbb{P}\, X \mid a \neq \varnothing\ \}$$

If X is a set, then $\mathbb{P}_1\, X$ is the set of all non-empty subsets of X.

B.3.7 Set union

function 30 leftassoc $(\_ \cup \_)$

$$[X]$$
$$\_ \cup \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X$$
$$\forall\, a, b : \mathbb{P}\, X \bullet a \cup b = \{\ x : X \mid x \in a \lor x \in b\ \}$$

The union of two sets of the same type is the set of values that are members of either set.

B.3.8 Set intersection

function 40 leftassoc $(\_ \cap \_)$

$$[X]$$
$$\_ \cap \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X$$
$$\forall\, a, b : \mathbb{P}\, X \bullet a \cap b = \{\ x : X \mid x \in a \land x \in b\ \}$$

The intersection of two sets of the same type is the set of values that are members of both sets.

B.3.9 Set difference

function 30 leftassoc $(\_ \setminus \_)$

$$[X]$$
$$\_ \setminus \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X$$
$$\forall\, a, b : \mathbb{P}\, X \bullet a \setminus b = \{\ x : X \mid x \in a \land x \notin b\ \}$$

The difference of two sets of the same type is the set of values that are members of the first set but not members of the second set.

B.3.10 Set symmetric difference

function 25 leftassoc $(\_ \ominus \_)$

$$[X]$$
$$\_ \ominus \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X$$
$$\forall\, a, b : \mathbb{P}\, X \bullet a \ominus b = \{\, x : X \mid \neg\, (x \in a \Leftrightarrow x \in b) \,\}$$

The symmetric set difference of two sets of the same type is the set of values that are members of one set, or the other, but not members of both.

B.3.11 Generalized union

$$[X]$$
$$\bigcup : \mathbb{P}\, \mathbb{P}\, X \to \mathbb{P}\, X$$
$$\forall\, A : \mathbb{P}\, \mathbb{P}\, X \bullet \bigcup A = \{\, x : X \mid \exists\, a : A \bullet x \in a \,\}$$

The generalized union of a set of sets of the same type is the set of values of that type that are members of at least one of the sets.

B.3.12 Generalized intersection

$$[X]$$
$$\bigcap : \mathbb{P}\, \mathbb{P}\, X \to \mathbb{P}\, X$$
$$\forall\, A : \mathbb{P}\, \mathbb{P}\, X \bullet \bigcap A = \{\, x : X \mid \forall\, a : A \bullet x \in a \,\}$$

The generalized intersection of a set of sets of values of the same type is the set of values of that type that are members of every one of the sets.

B.4 Finite sets

B.4.1 Finite subsets

generic $(\mathbb{F}\, \_)$

$$\mathbb{F}\, X == \bigcap \{\, A : \mathbb{P}\, \mathbb{P}\, X \mid \varnothing \in A \land (\forall\, a : A;\ x : X \bullet a \cup \{x\} \in A) \,\}$$

If X is a set, then $\mathbb{F}\, X$ is the set of all finite subsets of X. The set of finite subsets of X is the smallest set that contains the empty set and is closed under the action of adding single elements of X.

B.4.2 Non-empty finite subsets

$$\mathbb{F}_1\, X == \mathbb{F}\, X \setminus \{\varnothing\}$$

If X is a set, then $\mathbb{F}_1\, X$ is the set of all non-empty finite subsets of X. The set of non-empty finite subsets of X is the smallest set that contains the singleton sets of X and is closed under the action of adding single elements of X.

B.5 More notations for relations

section *relation_toolkit* parents *set_toolkit*

B.5.1 First component projection

$$[X, Y]$$
$$first : X \times Y \to X$$
$$\forall p : X \times Y \bullet first\ p = p.1$$

For any ordered pair p, $first\ p$ is the first component of the pair.

B.5.2 Second component projection

$$[X, Y]$$
$$second : X \times Y \to Y$$
$$\forall p : X \times Y \bullet second\ p = p.2$$

For any ordered pair p, $second\ p$ is the second component of the pair.

B.5.3 Maplet

function 10 leftassoc $(\_ \mapsto \_)$

$$[X, Y]$$
$$\_ \mapsto \_ : X \times Y \to X \times Y$$
$$\forall x : X;\ y : Y \bullet x \mapsto y = (x, y)$$

The maplet forms an ordered pair from two values; $x \mapsto y$ is just another notation for (x, y).

B.5.4 Domain

$$[X, Y]$$
$$dom : (X \leftrightarrow Y) \to \mathbb{P}\, X$$
$$\forall r : X \leftrightarrow Y \bullet dom\ r = \{\ p : r \bullet p.1\ \}$$

The domain of a relation r is the set of first components of the ordered pairs in r.

B.5.5 Range

$$[X, Y]$$
$$ran : (X \leftrightarrow Y) \to \mathbb{P}\, Y$$
$$\forall r : X \leftrightarrow Y \bullet ran\ r = \{\ p : r \bullet p.2\ \}$$

The range of a relation r is the set of second components of the ordered pairs in r.

B.5.6 Identity relation

generic $(id\ \_)$

$$id\ X == \{\ x : X \bullet x \mapsto x\ \}$$

The identity relation on a set X is the relation that relates every member of X to itself.

B.5.7 Relational composition

function 40 leftassoc $(\_ \fatsemi \_)$

$$[X, Y, Z]$$
$$\_ \fatsemi \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \to (X \leftrightarrow Z)$$
$$\forall r : X \leftrightarrow Y;\ s : Y \leftrightarrow Z \bullet r \fatsemi s = \{\ p : r;\ q : s \mid p.2 = q.1 \bullet p.1 \mapsto q.2\ \}$$

The relational composition of a relation $r : X \leftrightarrow Y$ and $s : Y \leftrightarrow Z$ is a relation of type $X \leftrightarrow Z$ formed by taking all the pairs p of r and q of s, where the second component of p is equal to the first component of q, and relating the first component of p with the second component of q.

B.5.8 Functional composition

function 40 leftassoc $(\_ \circ \_)$

$$[X, Y, Z]$$
$$\_ \circ \_ : (Y \leftrightarrow Z) \times (X \leftrightarrow Y) \to (X \leftrightarrow Z)$$
$$\forall r : X \leftrightarrow Y;\ s : Y \leftrightarrow Z \bullet s \circ r = r \fatsemi s$$

The functional composition of s and r is the same as the relational composition of r and s.

B.5.9 Domain restriction

function 65 rightassoc $(\_ \lhd \_)$

$$[X, Y]$$
$$\_ \lhd \_ : \mathbb{P}\, X \times (X \leftrightarrow Y) \to (X \leftrightarrow Y)$$
$$\forall a : \mathbb{P}\, X;\ r : X \leftrightarrow Y \bullet a \lhd r = \{\ p : r \mid p.1 \in a\ \}$$

The domain restriction of a relation $r : X \leftrightarrow Y$ by a set $a : \mathbb{P}\, X$ is the set of pairs in r whose first components are in a.

B.5.10 Range restriction

function 60 leftassoc $(\_ \rhd \_)$

$$[X, Y]$$
$$\_ \rhd \_ : (X \leftrightarrow Y) \times \mathbb{P}\, Y \to (X \leftrightarrow Y)$$
$$\forall r : X \leftrightarrow Y;\ b : \mathbb{P}\, Y \bullet r \rhd b = \{\ p : r \mid p.2 \in b\ \}$$

The range restriction of a relation $r : X \leftrightarrow Y$ by a set $b : \mathbb{P}\, Y$ is the set of pairs in r whose second components are in b.

B.5.11 Domain subtraction

function 65 rightassoc $(\_ \lhd \_)$

$$[X, Y]$$
$$\_ \lhd \_ : \mathbb{P}\, X \times (X \leftrightarrow Y) \to (X \leftrightarrow Y)$$
$$\forall a : \mathbb{P}\, X;\ r : X \leftrightarrow Y \bullet a \lhd r = \{\ p : r \mid p.1 \notin a\ \}$$

The domain subtraction of a relation $r : X \leftrightarrow Y$ by a set $a : \mathbb{P}\, X$ is the set of pairs in r whose first components are not in a.

B.5.12 Range subtraction

function 60 leftassoc $(\_ \rhd \_)$

$$[X, Y]$$
$$\_ \rhd \_ : (X \leftrightarrow Y) \times \mathbb{P}\, Y \to (X \leftrightarrow Y)$$
$$\forall r : X \leftrightarrow Y;\ b : \mathbb{P}\, Y \bullet r \rhd b = \{\ p : r \mid p.2 \notin b\ \}$$

The range subtraction of a relation $r : X \leftrightarrow Y$ by a set $b : \mathbb{P}\, Y$ is the set of pairs in r whose second components are not in b.

B.5.13 Relational inversion

function $(\_\ ^{\sim})$

$$[X, Y]$$
$$\_\ ^{\sim} : (X \leftrightarrow Y) \to (Y \leftrightarrow X)$$
$$\forall r : X \leftrightarrow Y \bullet r\ ^{\sim} = \{\ p : r \bullet p.2 \mapsto p.1\ \}$$

The inverse of a relation is the relation obtained by reversing every ordered pair in the relation.

B.5.14 Relational image

function $(\_ (\!| \_ |\!))$

$$[X, Y]$$
$$\_ (\!| \_ |\!) : (X \leftrightarrow Y) \times \mathbb{P}\, X \to \mathbb{P}\, Y$$
$$\forall r : X \leftrightarrow Y;\ a : \mathbb{P}\, X \bullet r(\!|\ a\ |\!) = \{\ p : r \mid p.1 \in a \bullet p.2\ \}$$

The relational image of a set $a : \mathbb{P}\, X$ through a relation $r : X \leftrightarrow Y$ is the set of values of type Y that are related under r to a value in a.

B.5.15 Overriding

function 50 leftassoc $(\_ \oplus \_)$

$$[X, Y]$$
$$\_ \oplus \_ : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \to (X \leftrightarrow Y)$$
$$\forall r, s : X \leftrightarrow Y \bullet r \oplus s = ((dom\ s) \lhd r) \cup s$$

If r and s are both relations between X and Y, the overriding of r by s is the whole of s together with those members of r that have no first components that are in the domain of s.

B.5.16 Transitive closure

function $(\_\ ^{+})$

$$[X]$$
$$\_\ ^{+} : (X \leftrightarrow X) \to (X \leftrightarrow X)$$
$$\forall r : X \leftrightarrow X \bullet r\ ^{+} = \bigcap \{\ s : X \leftrightarrow X \mid r \subseteq s \land r \fatsemi s \subseteq s\ \}$$

The transitive closure of a relation $r : X \leftrightarrow X$ is the smallest set that contains r and is closed under the action of composing r with its members.

B.5.17 Reflexive transitive closure

function $(\_^*)$

$$[X]$$
$$\_^* : (X \leftrightarrow X) \to (X \leftrightarrow X)$$
$$\forall r : X \leftrightarrow X \bullet r^* = r^+ \cup id\, X$$

The reflexive transitive closure of a relation $r : X \leftrightarrow X$ is the relation formed by extending the transitive closure of r by the identity relation on X.

B.6 Functions

section *function_toolkit* parents *relation_toolkit*

B.6.1 Partial functions

generic 5 rightassoc $(\_ \pfun \_)$

$$X \pfun Y == \{ f : X \leftrightarrow Y \mid \forall p, q : f \mid p.1 = q.1 \bullet p.2 = q.2 \}$$

$X \pfun Y$ is the set of all partial functions from X to Y, that is, the set of all relations between X and Y such that each x in X is related to at most one y in Y. The terms "function" and "partial function" are synonymous.

B.6.2 Partial injections

generic 5 rightassoc $(\_ \pinj \_)$

$$X \pinj Y == \{ f : X \leftrightarrow Y \mid \forall p, q : f \bullet p.1 = q.1 \Leftrightarrow p.2 = q.2 \}$$

$X \pinj Y$ is the set of partial injections from X to Y, that is, the set of all relations between X and Y such that each x in X is related to no more than one y in Y, and each y in Y is related to no more than one x in X. The terms "injection" and "partial injection" are synonymous.

B.6.3 Total injections

generic 5 rightassoc $(\_ \inj \_)$

$$X \inj Y == (X \pinj Y) \cap (X \to Y)$$

$X \inj Y$ is the set of total injections from X to Y, that is, the set of injections from X to Y that are also total functions from X to Y.

B.6.4 Partial surjections

generic 5 rightassoc $(\_ \psurj \_)$

$$X \psurj Y == \{ f : X \pfun Y \mid ran\, f = Y \}$$

$X \psurj Y$ is the set of partial surjections from X to Y, that is, the set of functions from X to Y whose range is equal to Y. The terms "surjection" and "partial surjection" are synonymous.

B.6.5 Total surjections

generic 5 rightassoc $(\_ \twoheadrightarrow \_)$

$$X \twoheadrightarrow Y == (X \nrightarrow Y) \cap (X \rightarrow Y)$$

$X \twoheadrightarrow Y$ is the set of total surjections from X to Y, that is, the set of surjections from X to Y that are also total functions from X to Y.

B.6.6 Bijections

generic 5 rightassoc $(\_ \rightarrowtail\!\!\!\!\!\twoheadrightarrow \_)$

$$X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y == (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$$

$X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y$ is the set of bijections from X to Y, that is, the set of total surjections from X to Y that are also total injections from X to Y.

B.6.7 Finite functions

generic 5 rightassoc $(\_ \nrightarrow\!\!\!\!\!\twoheadrightarrow \_)$

$$X \nrightarrow\!\!\!\!\!\twoheadrightarrow Y == (X \nrightarrow Y) \cap \mathbb{F}(X \times Y)$$

The finite functions from X to Y are the functions from X to Y that are also finite sets.

B.6.8 Finite injections

generic 5 rightassoc $(\_ \rightarrowtail\!\!\!\!\!\nrightarrow \_)$

$$X \rightarrowtail\!\!\!\!\!\nrightarrow Y == (X \nrightarrow\!\!\!\!\!\twoheadrightarrow Y) \cap (X \rightarrowtail Y)$$

The finite injections from X to Y are the injections from X to Y that are also finite functions from X to Y.

B.6.9 Disjointness

relation $(disjoint \_)$

$$[L, X]$$

$$disjoint \_ : \mathbb{P}(L \leftrightarrow \mathbb{P} X)$$

$$\forall f : L \leftrightarrow \mathbb{P} X \bullet disjoint\, f \Leftrightarrow (\forall p, q : f \mid p \neq q \bullet p.2 \cap q.2 = \varnothing)$$

A labelled family of sets is disjoint precisely when any distinct pair yields sets with no members in common.

B.6.10 Partitions

relation $(\_\ partition\ \_)$

$$[L, X]$$

$$\_\ partition\ \_ : (L \leftrightarrow \mathbb{P} X) \leftrightarrow \mathbb{P} X$$

$$\forall f : L \leftrightarrow \mathbb{P} X;\ a : \mathbb{P} X \bullet f\ partition\ a \Leftrightarrow disjoint\, f \land \bigcup(ran\, f) = a$$

A labelled family of sets f partitions a set a precisely when f is disjoint and the union of all the sets in f is a.

B.7 Numbers

section *number_toolkit*

B.7.1 Successor

function $(succ\ \_)$

$$succ\ \_ : \mathbb{P}(\mathbb{N} \times \mathbb{N})$$
$$(succ\ \_) = \lambda\, n : \mathbb{N} \bullet n + 1$$

The successor of a natural number n is equal to $n + 1$.

B.7.2 Integers

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

$\mathbb{Z}$ is the set of integers, that is, positive and negative whole numbers and zero. The set $\mathbb{Z}$ is characterised by axioms for its additive structure given in the prelude (clause 11) together with the next formal paragraph below.

Number systems that extend the integers may be specified as supersets of $\mathbb{Z}$.

B.7.3 Addition of integers, arithmetic negation

function $(\text{-}\ \_)$

$$\text{-}\ \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A})$$
$$\forall x, y : \mathbb{Z} \bullet \exists_1 z : \mathbb{Z} \bullet ((x, y), z) \in (\_ + \_)$$
$$\forall x : \mathbb{Z} \bullet \exists_1 y : \mathbb{Z} \bullet (x, y) \in (\text{-}\ \_)$$
$$\forall i, j, k : \mathbb{Z} \bullet$$
$$(i + j) + k = i + (j + k)$$
$$\land\ i + j = j + i$$
$$\land\ i + \text{-}\ i = 0$$
$$\land\ i + 0 = i$$
$$\mathbb{Z} = \{z : \mathbb{A} \mid \exists x : \mathbb{N} \bullet z = x \lor z = \text{-}\ x\}$$

The binary addition operator $(\_ + \_)$ is defined in the prelude (clause 11). The definition here introduces additional properties for integers. The addition and negation operations on integers are total functions that take integer values. The integers form a commutative group under $(\_ + \_)$ with $(\text{-}\ \_)$ as the inverse operation and 0 as the identity element.

NOTE If *function_toolkit* notation were exploited, the negation operator could be defined as follows.

$$\text{-}\ \_ : \mathbb{A} \pfun \mathbb{A}$$
$$(\mathbb{Z} \times \mathbb{Z}) \lhd (\_ + \_) \in \mathbb{Z} \times \mathbb{Z} \fun \mathbb{Z}$$
$$\mathbb{Z} \lhd (\text{-}\ \_) \in \mathbb{Z} \fun \mathbb{Z}$$
$$\forall i, j, k : \mathbb{Z} \bullet$$
$$(i + j) + k = i + (j + k)$$
$$\land\ i + j = j + i$$
$$\land\ i + \text{-}\ i = 0$$
$$\land\ i + 0 = i$$
$$\forall h : \mathbb{P}\, \mathbb{Z} \bullet$$
$$1 \in h \land (\forall i, j : h \bullet i + j \in h \land \text{-}\ i \in h)$$
$$\Rightarrow h = \mathbb{Z}$$

B.7.4 Subtraction

function 30 leftassoc $(\_ - \_)$

$$\begin{array}{|l} \_ - \_ : \mathbb{P}((\mathbb{A} \times \mathbb{A}) \times \mathbb{A}) \\ \hline \forall x, y : \mathbb{Z} \bullet \exists_1 z : \mathbb{Z} \bullet ((x, y), z) \in (\_ - \_) \\ \forall i, j : \mathbb{Z} \bullet i - j = i +\!\!- j \end{array}$$

Subtraction is a function whose domain includes all pairs of integers. For all integers i and j, $i - j$ is equal to $i +\!\!- j$.

NOTE If *function_toolkit* notation were exploited, the subtraction operator could be defined as follows.

$$\begin{array}{|l} \_ - \_ : \mathbb{A} \times \mathbb{A} \pfun \mathbb{A} \\ \hline (\mathbb{Z} \times \mathbb{Z}) \lhd (\_ - \_) \in \mathbb{Z} \times \mathbb{Z} \fun \mathbb{Z} \\ \forall i, j : \mathbb{Z} \bullet i - j = i +\!\!- j \end{array}$$

B.7.5 Less-than-or-equal

relation $(\_ \leq \_)$

$$\begin{array}{|l} \_ \leq \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A}) \\ \hline \forall i, j : \mathbb{Z} \bullet i \leq j \Leftrightarrow j - i \in \mathbb{N} \end{array}$$

For all integers i and j, $i \leq j$ if and only if their difference $j - i$ is a natural number.

B.7.6 Less-than

relation $(\_ < \_)$

$$\begin{array}{|l} \_ < \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A}) \\ \hline \forall i, j : \mathbb{Z} \bullet i < j \Leftrightarrow i + 1 \leq j \end{array}$$

For all integers i and j, $i < j$ if and only if $i + 1 \leq j$.

B.7.7 Greater-than-or-equal

relation $(\_ \geq \_)$

$$\begin{array}{|l} \_ \geq \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A}) \\ \hline \forall i, j : \mathbb{Z} \bullet i \geq j \Leftrightarrow j \leq i \end{array}$$

For all integers i and j, $i \geq j$ if and only if $j \leq i$.

B.7.8 Greater-than

relation $(\_ > \_)$

$$\begin{array}{|l} \_ > \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A}) \\ \hline \forall i, j : \mathbb{Z} \bullet i > j \Leftrightarrow j < i \end{array}$$

For all integers i and j, $i > j$ if and only if $j < i$.

B.7.9 Strictly positive natural numbers

$$\mathbb{N}_1 == \{x : \mathbb{N} \mid \neg\, x = 0\}$$

The strictly positive natural numbers $\mathbb{N}_1$ are the natural numbers except zero.

B.7.10 Non-zero integers

$$\mathbb{Z}_1 == \{x : \mathbb{Z} \mid \neg\, x = 0\}$$

The non-zero integers $\mathbb{Z}_1$ are the integers except zero.

B.7.11 Multiplication of integers

function 40 leftassoc $(\_ * \_)$

$$\_ * \_ : \mathbb{P}((\mathbb{A} \times \mathbb{A}) \times \mathbb{A})$$

$$\begin{aligned}
&\forall\, x, y : \mathbb{Z} \bullet \exists_1\, z : \mathbb{Z} \bullet ((x, y), z) \in (_ * _)\\
&\forall\, i, j, k : \mathbb{Z} \bullet\\
&\quad (i * j) * k = i * (j * k)\\
&\quad \land\ i * j = j * i\\
&\quad \land\ i * (j + k) = i * j + i * k\\
&\quad \land\ 0 * i = 0\\
&\quad \land\ 1 * i = i
\end{aligned}$$

The binary multiplication operator $(\_ * \_)$ is defined for integers. The multiplication operation on integers is a total function and has integer values. Multiplication on integers is characterised by the unique operation under which the integers become a commutative ring with identity element 1.

NOTE If *function_toolkit* notation were exploited, the multiplication operator could be defined as follows.

$$\_ * \_ : (\mathbb{A} \times \mathbb{A}) \pfun \mathbb{A}$$

$$\begin{aligned}
&(\mathbb{Z} \times \mathbb{Z}) \domres (_ * _) \in \mathbb{Z} \times \mathbb{Z} \pfun \mathbb{Z}\\
&\forall\, i, j, k : \mathbb{Z} \bullet\\
&\quad (i * j) * k = i * (j * k)\\
&\quad \land\ i * j = j * i\\
&\quad \land\ i * (j + k) = i * j + i * k\\
&\quad \land\ 0 * i = 0\\
&\quad \land\ 1 * i = i
\end{aligned}$$

B.7.12 Division, modulus

function 40 leftassoc $(\_\ div\ \_)$
function 40 leftassoc $(\_\ mod\ \_)$

$$\_\ div\ \_\ ,\_\ mod\ \_ : \mathbb{P}((\mathbb{A} \times \mathbb{A}) \times \mathbb{A})$$

$$\begin{aligned}
&\forall\, x : \mathbb{Z};\ y : \mathbb{Z}_1 \bullet \exists_1\, z : \mathbb{Z} \bullet ((x, y), z) \in (_\ div\ _)\\
&\forall\, x : \mathbb{Z};\ y : \mathbb{Z}_1 \bullet \exists_1\, z : \mathbb{Z} \bullet ((x, y), z) \in (_\ mod\ _)\\
&\forall\, i : \mathbb{Z};\ j : \mathbb{Z}_1 \bullet\\
&\quad i = (i\ div\ j) * j + i\ mod\ j\\
&\quad \land\ (0 \leq i\ mod\ j < j \lor j < i\ mod\ j \leq 0)
\end{aligned}$$

For all integers i and non-zero integers j, the pair (i, j) is in the domain of $\_div\_$ and of $\_mod\_$, and $i \mathit{div} j$ and $i \mathit{mod} j$ have integer values.

When not zero, $i \mathit{mod} j$ has the same sign as j. This means that $i \mathit{div} j$ is the largest integer no greater than the rational number i/j.

NOTE If *function_toolkit* notation were exploited, the division and modulus operators could be defined as follows.

$$\_ \mathit{div} \_ , \_ \mathit{mod} \_ : \mathbb{A} \times \mathbb{A} \pfun \mathbb{A}$$

$$(\mathbb{Z} \times \mathbb{Z}_1) \dres (\_ \mathit{div} \_) \in \mathbb{Z} \times \mathbb{Z}_1 \fun \mathbb{Z}$$

$$(\mathbb{Z} \times \mathbb{Z}_1) \dres (\_ \mathit{mod} \_) \in \mathbb{Z} \times \mathbb{Z}_1 \fun \mathbb{Z}$$

$$\forall i : \mathbb{Z}; \ j : \mathbb{Z}_1 \bullet i = (i \mathit{div} j) * j + i \mathit{mod} j \land (0 \leq i \mathit{mod} j < j \lor j < i \mathit{mod} j \leq 0)$$

B.8 Sequences

section *sequence_toolkit* parents *function_toolkit*, *number_toolkit*

B.8.1 Number range

function 20 leftassoc $(\_ \upto \_)$

$$\_ \upto \_ : \mathbb{A} \times \mathbb{A} \pfun \mathbb{P} \mathbb{A}$$

$$(\mathbb{Z} \times \mathbb{Z}) \dres (\_ \upto \_) \in \mathbb{Z} \times \mathbb{Z} \fun \mathbb{P} \mathbb{Z}$$

$$\forall i, j : \mathbb{Z} \bullet i \upto j = \{ k : \mathbb{Z} \mid i \leq k \leq j \}$$

The number range from i to j is the set of all integers greater than or equal to i, which are also less than or equal to j.

B.8.2 Iteration

$$[X]$$

$$\mathit{iter} : \mathbb{Z} \fun (X \rel X) \fun (X \rel X)$$

$$\forall r : X \rel X \bullet \mathit{iter}\ 0\ r = \mathit{id}\ X$$

$$\forall r : X \rel X; \ n : \mathbb{N} \bullet \mathit{iter}\ (n + 1)\ r = r \fatsemi (\mathit{iter}\ n\ r)$$

$$\forall r : X \rel X; \ n : \mathbb{N} \bullet \mathit{iter}\ (-\ n)\ r = \mathit{iter}\ n\ (r^{\sim})$$

iter is the iteration function for a relation. The iteration of a relation $r : X \rel X$ for zero times is the identity relation on X. The iteration of a relation $r : X \rel X$ for $n + 1$ times is the composition of the relation with its iteration n times. The iteration of a relation $r : X \rel X$ for - n times is the iteration for n times of the inverse of the relation.

function $(\_^{-})$

$$[X]$$

$$\_^{-} : (X \rel X) \times \mathbb{Z} \fun (X \rel X)$$

$$\forall r : X \rel X; \ n : \mathbb{N} \bullet r^{n} = \mathit{iter}\ n\ r$$

iter n r may be written as r^{n}.

B.8.3 Number of members of a set

function $(\# \_)$

$$[X]$$
$$\# \_ : \mathbb{F}\, X \to \mathbb{N}$$
$$\forall\, a : \mathbb{F}\, X \bullet \# a = (\mu\, n : \mathbb{N} \mid (\exists f : 1 \mathinner{..} n \rightarrowtail a \bullet ran\, f = a))$$

The number of members of a finite set is the upper limit of the number range starting at 1 that can be put into bijection with the set.

B.8.4 Minimum

function $(min \_)$

$$min \_ : \mathbb{P}\, \mathbb{A} \pfun \mathbb{A}$$
$$\mathbb{P}\, \mathbb{Z} \lhd (min \_) = \{\ a : \mathbb{P}\, \mathbb{Z};\ m : \mathbb{Z} \mid m \in a \land (\forall\, n : a \bullet m \leq n) \bullet a \mapsto m\ \}$$

If a set of integers has a member that is less than or equal to all members of that set, that member is its minimum.

B.8.5 Maximum

function $(max \_)$

$$max \_ : \mathbb{P}\, \mathbb{A} \pfun \mathbb{A}$$
$$\mathbb{P}\, \mathbb{Z} \lhd (max \_) = \{\ a : \mathbb{P}\, \mathbb{Z};\ m : \mathbb{Z} \mid m \in a \land (\forall\, n : a \bullet n \leq m) \bullet a \mapsto m\ \}$$

If a set of integers has a member that is greater than or equal to all members of that set, that member is its maximum.

B.8.6 Finite sequences

generic $(seq \_)$

$$seq\, X == \{\ f : \mathbb{N} \pfun X \mid dom\, f = 1 \mathinner{..} \# f\ \}$$

A finite sequence is a finite indexed set of values of the same type, whose domain is a contiguous set of positive integers starting at 1.

$seq\, X$ is the set of all finite sequences of values of X, that is, of finite functions from the set $1 \mathinner{..} n$, for some n, to elements of X.

B.8.7 Non-empty finite sequences

$$seq_1\, X == seq\, X \setminus \{\varnothing\}$$

$seq_1\, X$ is the set of all non-empty finite sequences of values of X.

B.8.8 Injective sequences

generic $(iseq \_)$

$$iseq\, X == seq\, X \cap (\mathbb{N} \pinj X)$$

$iseq\, X$ is the set of all injective finite sequences of values of X, that is, of finite sequences over X that are also injections.

B.8.9 Sequence brackets

function $(\langle \ , , \ \rangle)$

$$\langle \ , , \ \rangle[X] == \lambda\, s : seq\ X \bullet s$$

The brackets $\langle$ and $\rangle$ can be used for enumerated sequences.

B.8.10 Concatenation

function 30 leftassoc $(\_ \frown \_)$

$$[X]$$
$$\_ \frown \_ : seq\ X \times seq\ X \to seq\ X$$
$$\forall\, s, t : seq\ X \bullet s \frown t = s \cup \{\ n : dom\ t \bullet n + \#s \mapsto t\ n\ \}$$

Concatenation is a function of a pair of finite sequences of values of the same type whose result is a sequence that begins with all elements of the first sequence and continues with all elements of the second sequence.

B.8.11 Reverse

$$[X]$$
$$rev : seq\ X \to seq\ X$$
$$\forall\, s : seq\ X \bullet rev\ s = \lambda\, n : dom\ s \bullet s(\#s - n + 1)$$

The reverse of a sequence is the sequence obtained by taking its elements in the opposite order.

B.8.12 Head of a sequence

$$[X]$$
$$head : seq_1\ X \to X$$
$$\forall\, s : seq_1\ X \bullet head\ s = s\ 1$$

If s is a non-empty sequence of values, then $head\ s$ is the value that is first in the sequence.

B.8.13 Last of a sequence

$$[X]$$
$$last : seq_1\ X \to X$$
$$\forall\, s : seq_1\ X \bullet last\ s = s(\#s)$$

If s is a non-empty sequence of values, then $last\ s$ is the value that is last in the sequence.

B.8.14 Tail of a sequence

$$[X]$$
$$tail : seq_1\ X \to seq\ X$$
$$\forall\, s : seq_1\ X \bullet tail\ s = \lambda\, n : 1\,..\,(\#s - 1) \bullet s(n + 1)$$

If s is a non-empty sequence of values, then $tail\ s$ is the sequence of values that is obtained from s by discarding the first element and renumbering the remainder.

B.8.15 Front of a sequence

$$[X]$$
$$front : seq_1 X \to seq X$$
$$\forall s : seq_1 X \bullet front\ s = \{\#s\} \lhd s$$

If s is a non-empty sequence of values, then $front\ s$ is the sequence of values that is obtained from s by discarding the last element.

B.8.16 Squashing

$$[X]$$
$$squash : (\mathbb{Z} \pfun X) \to seq X$$
$$\forall f : \mathbb{Z} \pfun X \bullet squash\ f = \{\ p : f \bullet \#\{\ i : dom\ f \mid i \leq p.1\ \} \mapsto p.2\ \}$$

squash takes a finite function $f : \mathbb{Z} \pfun X$ and renumbers its domain to produce a finite sequence.

B.8.17 Extraction

function 45 rightassoc $(\_ \upharpoonright \_)$

$$[X]$$
$$\_ \upharpoonright \_ : \mathbb{P}\,\mathbb{Z} \times seq\ X \to seq\ X$$
$$\forall a : \mathbb{P}\,\mathbb{Z};\ s : seq\ X \bullet a \upharpoonright s = squash(a \lhd s)$$

The extraction of a set a of indices from a sequence is the sequence obtained from the original by discarding any indices that are not in the set a, then renumbering the remainder.

B.8.18 Filtering

function 40 leftassoc $(\_ \restriction \_)$

$$[X]$$
$$\_ \restriction \_ : seq\ X \times \mathbb{P}\ X \to seq\ X$$
$$\forall s : seq\ X;\ a : \mathbb{P}\ X \bullet s \restriction a = squash(s \rhd a)$$

The filter of a sequence by a set a is the sequence obtained from the original by discarding any members that are not in the set a, then renumbering the remainder.

B.8.19 Prefix relation

relation $(\_\ prefix\ \_)$

$$[X]$$
$$\_\ prefix\ \_ : seq\ X \leftrightarrow seq\ X$$
$$\forall s, t : seq\ X \bullet s\ prefix\ t \Leftrightarrow s \subseteq t$$

A sequence s is a prefix of another sequence t if it forms the front portion of t.

B.8.20 Suffix relation

relation $(\_\ suffix\ \_)$

$$[X]$$
$$\_\ suffix\ \_ : seq\ X \leftrightarrow seq\ X$$
$$\forall\ s, t : seq\ X \bullet s\ suffix\ t \Leftrightarrow (\exists\ u : seq\ X \bullet u \frown s = t)$$

A sequence s is a suffix of another sequence t if it forms the end portion of t.

B.8.21 Infix relation

relation $(\_\ infix\ \_)$

$$[X]$$
$$\_\ infix\ \_ : seq\ X \leftrightarrow seq\ X$$
$$\forall\ s, t : seq\ X \bullet s\ infix\ t \Leftrightarrow (\exists\ u, v : seq\ X \bullet u \frown s \frown v = t)$$

A sequence s is an infix of another sequence t if it forms a mid portion of t.

B.8.22 Distributed concatenation

$$[X]$$
$$\frown/ : seq\ seq\ X \rightarrow seq\ X$$
$$\frown/\ \langle\ \rangle = \langle\ \rangle$$
$$\forall\ s : seq\ X \bullet \frown/\ \langle s \rangle = s$$
$$\forall\ q, r : seq\ seq\ X \bullet \frown/(q \frown r) = (\frown/\ q) \frown (\frown/\ r)$$

The distributed concatenation of a sequence t of sequences of values of type X is the sequence of values of type X that is obtained by concatenating the members of t in order.

B.9 Standard toolkit

section *standard_toolkit* parents *sequence_toolkit*

The standard toolkit contains the definitions of section *sequence_toolkit* (and implicitly those of its ancestral sections).

Annex C
(normative)

Organisation by concrete syntax production

C.1 Introduction

This annex duplicates some of the definitions presented in the normative clauses, but re-organised by concrete syntax production. This re-organisation provides no suitable place to accommodate the material listed in the rest of this introduction. That material is consequently omitted from this annex.

a) From Concrete syntax, the rules defining:

   1) `Formals`, used in Generic axiomatic description paragraph, Generic schema paragraph, Generic horizontal definition paragraph, and Generic conjecture paragraph;

   2) `DeclName`, used in `Branch`, Schema hiding expression, Schema renaming expression, Colon declaration and Equal declaration;

   3) `RefName`, used in Reference expression, Generic instantiation expression, and Binding selection expression;

   4) `OpName` and its auxiliaries, used in `RefName` and `DeclName`;

   5) `ExpSep` and `ExpressionList`, used in auxiliaries of Relation operator application predicates and Function and generic operator application expressions;

   6) and also the operator precedences and associativities and additional syntactic restrictions.

b) From Characterisation rules:

   1) Characteristic tuple.

c) From Prelude:

   1) its text is relevant not just to number literal expressions but also to the list arguments in Relation operator application predicates and Function and generic operator application expressions.

d) From Syntactic transformation rules:

   1) `Name` and `ExpressionList`.

e) From Type inference rules:

   1) Generic type instantiation, carrier set and implicit instantiation.

   2) Summary of scope rules.

f) From Semantic relations:

   1) all the relations for `Type` are omitted.

Also, the description of the overall effect of a phase, or how the phase operates, is generally omitted from this annex.

Moreover, some of the phases and representations are entirely omitted here, namely Mark-ups, Z characters, Lexis and Annotated syntax.

C.2 Specification

C.2.1 Introduction

`Specification` is the start symbol of the syntax. A `Specification` can be either a sectioned specification or an anonymous specification. A sectioned specification comprises a sequence of named sections. An anonymous specification comprises a single anonymous section.

C.2.2 Sectioned specification

C.2.2.1 Syntax

```
Specification     = { Section }
                  | ...
                  ;
```

C.2.2.2 Type

$$\frac{\{\} \vdash^{\mathcal{S}} s_{prelude} \fatsemi \Gamma_0 \qquad \delta_1 \vdash^{\mathcal{S}} s_1 \fatsemi \Gamma_1 \qquad \ldots \qquad \delta_n \vdash^{\mathcal{S}} s_n \fatsemi \Gamma_n}{\vdash^{\mathcal{Z}} s_1 \fatsemi \Gamma_1 \ldots s_n \fatsemi \Gamma_n} \begin{pmatrix} \delta_1 = \{prelude \mapsto \Gamma_0\} \\ \vdots \\ \delta_n = \delta_{n-1} \cup \{i_{n-1} \mapsto \Gamma_{n-1}\} \end{pmatrix}$$

where i_{n-1} is the name of section s_{n-1}, and none of the sections $s_1 \ldots s_n$ are named *prelude*.

Each section is typechecked in an environment formed from preceding sections, and is annotated with an environment that it establishes.

NOTE The section-type environment established by the prelude section is as follows.

$$\begin{aligned} \Gamma_0 \quad = \quad & (\mathbb{A}, (prelude, \mathbb{P}(\texttt{GIVEN}\ \mathbb{A}))); \\ & (\mathbb{N}, (prelude, \mathbb{P}(\texttt{GIVEN}\ \mathbb{A}))); \\ & (number\_literal\_0, (prelude, (\texttt{GIVEN}\ \mathbb{A}))); \\ & (number\_literal\_1, (prelude, (\texttt{GIVEN}\ \mathbb{A}))); \\ & (\bowtie\!+\!\bowtie, (prelude, \mathbb{P}(((\texttt{GIVEN}\ \mathbb{A}) \times (\texttt{GIVEN}\ \mathbb{A})) \times (\texttt{GIVEN}\ \mathbb{A})))) \end{aligned}$$

If one of the sections $s_1 \ldots s_n$ is named *prelude*, then the same type inference rule applies except that the type subsequent for the prelude section is omitted.

C.2.2.3 Semantics

$$[\![\ s_1 \ldots s_n\ ]\!]^{\mathcal{Z}} \quad = \quad ([\![\ \texttt{ZED}\ \text{section}\ prelude\ \text{parents}\ \texttt{END} \ldots\ ]\!]^{\mathcal{S}} \fatsemi [\![\ s_1\ ]\!]^{\mathcal{S}} \fatsemi \ldots \fatsemi [\![\ s_n\ ]\!]^{\mathcal{S}})\ \varnothing$$

The meaning of the Z specification $s_1 \ldots s_n$ is the function from sections' names to their sets of models formed by starting with the empty function and extending that with a maplet from a section's name to its set of models for each section in the specification, starting with the prelude.

To determine $[\![\ \texttt{ZED}\ \text{section}\ prelude\ \text{parents}\ \texttt{END} \ldots\ ]\!]^{\mathcal{Z}}$ another prelude shall not be prefixed onto it.

NOTE The meaning of a specification is not the meaning of its last section, so as to permit several meaningful units within a single document.

C.2.3 Anonymous specification

C.2.3.1 Syntax

```
Specification     = ...
                  | { Paragraph }
                  ;
```

C.2.3.2 Transformation

The anonymous specification D_1 ... D_n is semantically equivalent to the sectioned specification comprising a single section containing those paragraphs with the mathematical toolkit of annex B as its parent.

$$D_1 \ldots D_n \implies \textit{Mathematical toolkit}\ \texttt{ZED}\ \text{section}\ \textit{Specification}\ \text{parents}\ \textit{standard\_toolkit}\ \texttt{END}\ D_1 \ldots D_n$$

In this transformation, *Mathematical toolkit* denotes the entire text of annex B. The name given to the section is not important: it need not be *Specification*, though it shall not be *prelude* or that of any section of the mathematical toolkit.

C.3 Section

C.3.1 Introduction

A `Section` can be either an inheriting section or a base section. An inheriting section gathers together the paragraphs of parent sections with new paragraphs. A base section is like an inheriting section but has no parents.

C.3.2 Inheriting section

C.3.2.1 Syntax

```
Section           = ZED , section , NAME , parents , [ NAME , { ,-tok , NAME } ] , END ,
                        { Paragraph }
                  | ...
                  ;
```

C.3.2.2 Type

$$\frac{\beta_0 \vdash^{\mathcal{D}} D_1 \mathbin{⨟} \sigma_1 \quad \ldots \quad \beta_{n-1} \vdash^{\mathcal{D}} D_n \mathbin{⨟} \sigma_n}{\Lambda \vdash^{\mathcal{S}} \texttt{ZED}\ \text{section}\ i \ \text{parents}\ i_1, \ldots, i_m\ \texttt{END} \ D_1 \mathbin{⨟} \sigma_1 \ \ldots\ D_n \mathbin{⨟} \sigma_n \mathbin{⨟} \Gamma}$$

$$\left(\begin{array}{l} i \notin dom\ \Lambda \\ \{i_1, \ldots, i_m\} \subseteq dom\ \Lambda \\ \gamma_{-1} = \text{if}\ i = \textit{prelude}\ \text{then}\ \{\}\ \text{else}\ \Lambda\ \textit{prelude} \\ \gamma_0 = \gamma_{-1} \cup \Lambda\ i_1 \cup \ldots \cup \Lambda\ i_m \\ \beta_0 = \gamma_0 \mathbin{⨟} \textit{second} \\ \textit{disjoint}\ \langle dom\ \sigma_1, \ldots, dom\ \sigma_n \rangle \\ \Gamma \in (\_ \pfun \_) \\ \Gamma = \gamma_0 \cup \{j : \texttt{NAME};\ \tau : \texttt{Type} \mid j \mapsto \tau \in \sigma_1 \cup \ldots \cup \sigma_n \bullet j \mapsto (i, \tau)\} \\ \beta_1 = \beta_0 \cup \sigma_1 \\ \quad \vdots \\ \beta_{n-1} = \beta_{n-2} \cup \sigma_{n-1} \end{array}\right)$$

Taking the side-conditions in order, this type inference rule ensures that:

a) the name of the section, i, is different from that of any previous section;

b) the names in the parents list are names of known sections;

c) the section environment of the prelude is included if the section is not itself the prelude;

d) the section-type environment γ_o is formed from those of the parents;

e) the type environment β_o is determined from the section-type environment γ_o;

f) there is no global redefinition between any pair of paragraphs of the section (the sets of names in their signatures are disjoint);

g) a name which is common to the environments of multiple parents shall have originated in a common ancestral section, and a name introduced by a paragraph of this section shall not also be introduced by another paragraph or parent section (all ensured by the combined environment being a function);

h) the annotation of the section is an environment formed from those of its parents extended according to the signatures of its paragraphs;

i) and the type environment in which a paragraph is typechecked is formed from that of the parent sections extended with the signatures of the preceding paragraphs of this section.

NOTE 1 Ancestors need not be immediate parents, and a section cannot be amongst its own ancestors (no cycles in the parent relation).

NOTE 2 The name of a section can be the same as the name of a variable introduced in a declaration—the two are not confused.

C.3.2.3 Semantics

The prelude section, as defined in clause 11, is treated specially, as it is the only one that does not have prelude as an implicit parent.

$$[\![\texttt{ZED section } \mathit{prelude} \texttt{ parents END } D_1 \ldots D_n]\!]^{\mathcal{S}} = \lambda\ T : \mathit{SectionModels} \bullet \{\mathit{prelude} \mapsto ([\![D_1]\!]^{\mathcal{D}} \fatsemi \ldots \fatsemi [\![D_n]\!]^{\mathcal{D}})\ (\!|\ \{\varnothing\}\ |\!)\}$$

The meaning of the prelude section is given by that constant function which, whatever function from sections' names and their sets of models it is given, returns the singleton set mapping the name *prelude* to its set of models. The set of models is that to which the set containing an empty model is related by the composition of the relations between models that denote the meanings of each of the prelude's paragraphs—see clause 11 for details of those paragraphs.

NOTE One model of the prelude section can be written as follows.

$\{\mathbb{A} \mapsto \mathbb{A},$
$\mathbb{N} \mapsto \mathbb{N},$
$\mathit{number\_literal\_0} \mapsto 0,$
$\mathit{number\_literal\_1} \mapsto 1,$
$\_ + \_ \mapsto \{((0,0),0),((0,1),1),((1,0),1),((1,1),2),\ldots\}\}$

The behaviour of $(\_ + \_)$ on non-natural numbers, e.g. reals, has not been defined at this point, so the set of models for the prelude section includes alternatives for every possible extended behaviour of addition.

$$[\![\texttt{ZED section } i \texttt{ parents } i_1, \ldots, i_m \texttt{ END } D_1 \ldots D_n]\!]^{\mathcal{S}} = \lambda\ T : \mathit{SectionModels} \bullet T \cup \{i \mapsto ([\![D_1]\!]^{\mathcal{D}} \fatsemi \ldots \fatsemi [\![D_n]\!]^{\mathcal{D}})\ (\!|\ \{M_o : T\ \mathit{prelude};\ M_1 : T\ i_1;\ \ldots;\ M_m : T\ i_m;\ M : \mathit{Model} \mid M = M_o \cup M_1 \cup \ldots \cup M_m \bullet M\}\ |\!)\}$$

The meaning of a section other than the prelude is the extension of a function from sections' names to their sets of models with a maplet from the given section's name to its set of models. The given section's set of models is that to which the union of the models of the section's parents is related by the composition of the relations between models that denote the meanings of each of the section's paragraphs.

C.3.3 Base section

C.3.3.1 Syntax

```
Section          = ...
                 | ZED , section , NAME , END , { Paragraph }
                 ;
```

C.3.3.2 Transformation

The base section ZED section i END D_1 ... D_n is semantically equivalent to the inheriting section that inherits from no parents (bar *prelude*).

$$\texttt{ZED section } i \texttt{ END } D_1 \ldots D_n \implies \texttt{ZED section } i \texttt{ parents END } D_1 \ldots D_n$$

C.4 Paragraph

C.4.1 Introduction

A Paragraph can introduce new names into the models, and can constrain the values associated with names. A Paragraph can be any of given types, axiomatic description, schema definition, generic axiomatic description, generic schema definition, horizontal definition, generic horizontal definition, generic operator definition, free types, conjecture, generic conjecture, or operator template.

C.4.2 Given types

C.4.2.1 Syntax

```
Paragraph        = ZED , [-tok , NAME , { ,-tok , NAME } , ]-tok , END
                 | ...
                 ;
```

C.4.2.2 Type

$$\frac{}{\Sigma \vdash^{\mathcal{P}} \texttt{ZED } [i_1, \ldots, i_n] \texttt{ END} \ \wp\ \sigma} \left(\begin{array}{l} \#\{i_1, \ldots, i_n\} = n \\ \sigma = i_1 : \mathbb{P}(\texttt{GIVEN } i_1); \ \ldots; \ i_n : \mathbb{P}(\texttt{GIVEN } i_n) \end{array} \right)$$

In a given types paragraph, there shall be no duplication of names. The annotation of the paragraph is a signature associating the given type names with powerset types.

C.4.2.3 Semantics

The given types paragraph ZED $[i_1, \ldots, i_n]$ END introduces unconstrained global names.

$$\begin{array}{rcl} [\![\texttt{ZED } [i_1, \ldots, i_n] \texttt{ END}]\!]^{\mathcal{D}} & = & \{M : Model; \ w_1, \ \ldots, \ w_n : \mathbb{W} \\ & & \bullet\ M \mapsto M \cup \{i_1 \mapsto w_1, \ \ldots, \ i_n \mapsto w_n\} \\ & & \quad \cup \{i_1\ decor\ \heartsuit \mapsto w_1, \ \ldots, \ i_n\ decor\ \heartsuit \mapsto w_n\}\} \end{array}$$

It relates a model M to that model extended with associations between the names of the given types and semantic values chosen to represent their carrier sets. Associations for names decorated with the reserved stroke ♡ are also introduced, so that references to them from given types (see 15.2.6.1) can avoid being captured.

C.4.3 Axiomatic description

C.4.3.1 Syntax

```
Paragraph       = ...
                | AX , SchemaText , END
                | ...
                ;
```

C.4.3.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \ ⦂ \ \tau}{\Sigma \vdash^{\mathcal{D}} \texttt{AX}\ e \ ⦂ \ \tau\ \texttt{END} \ ⦂ \ \sigma}(\tau = \mathbb{P}[\sigma])$$

In an axiomatic description paragraph `AX` e `END`, the expression e shall be a schema. The annotation of the paragraph is the signature of that schema.

C.4.3.3 Semantics

The axiomatic description paragraph `AX` e `END` introduces global names and constraints on their values.

$$[\![\ \texttt{AX}\ e\ \texttt{END} \ ]\!]^{\mathcal{D}} \quad = \quad \{M : \mathit{Model};\ t : \mathbb{W} \mid t \in [\![\ e \ ]\!]^{\mathcal{E}} M \bullet M \mapsto M \cup t\}$$

It relates a model M to that model extended with a binding t of the schema that is the value of e in model M.

C.4.4 Schema definition

C.4.4.1 Syntax

```
Paragraph       = ...
                | SCH , NAME , SchemaText , END
                | ...
                ;
```

C.4.4.2 Transformation

The schema definition paragraph `SCH` i t `END` introduces the global name i, associating it with the schema that is the value of t.

$$\texttt{SCH}\ i\ t\ \texttt{END} \quad \Longrightarrow \quad \texttt{AX}\ [i == t]\ \texttt{END}$$

The paragraph is semantically equivalent to the axiomatic description paragraph whose sole declaration associates the schema's name with the expression resulting from syntactic transformation of the schema text.

C.4.5 Generic axiomatic description

C.4.5.1 Syntax

```
Paragraph       = ...
                | GENAX , [-tok , Formals , ]-tok , SchemaText , END
                | ...
                ;
```

C.4.5.2 Type

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_n)\} \vdash^{\mathcal{E}} e\ \wr\ \tau}{\Sigma \vdash^{\mathcal{D}} \texttt{GENAX}\ [i_1, ..., i_n]\ e\ \wr\ \tau\ \texttt{END}\ \wr\ \sigma} \left(\begin{array}{l} \#\{i_1,\ ...,\ i_n\} = n \\ \tau = \mathbb{P}[\beta] \\ \sigma = \lambda\ j : dom\ \beta \bullet [i_1, ..., i_n]\ (\beta\ j) \end{array} \right)$$

In a generic axiomatic description paragraph GENAX $[i_1, ..., i_n]$ e END, there shall be no duplication of names within the generic parameters. The expression e is typechecked, in an environment overridden by the generic parameters, and shall be a schema. The annotation of the paragraph is formed from the signature of that schema, having the same names but associated with types that are generic.

C.4.5.3 Semantics

The generic axiomatic description paragraph GENAX $[i_1, ..., i_n]$ e END introduces global names and constraints on their values, with generic parameters that have to be instantiated (by sets) whenever those names are referenced.

$$\begin{array}{l} [\![\ \texttt{GENAX}\ [i_1, ..., i_n]\ (e\ \wr\ \mathbb{P}[j_1 : \tau_1;\ ...;\ j_m : \tau_m])\ \texttt{END}\ ]\!]^{\mathcal{D}} = \\ \quad \{M : Model;\ u : \mathbb{W} \upharpoonright n \rightarrow \mathbb{W} \\ \quad\quad \mid \forall\ w_1,\ ...,\ w_n : \mathbb{W} \bullet \exists\ w : \mathbb{W} \bullet \\ \quad\quad\quad u\ (w_1,\ ...,\ w_n) \in w \\ \quad\quad\quad\quad \wedge\ (M \oplus \{i_1 \mapsto w_1,\ ...,\ i_n \mapsto w_n\} \cup \{i_1\ decor\ \spadesuit \mapsto w_1,\ ...,\ i_n\ decor\ \spadesuit \mapsto w_n\}) \mapsto w \in [\![\ e\ ]\!]^{\mathcal{E}} \\ \quad\quad \bullet\ M \mapsto M \cup \lambda\ y : \{j_1,\ ...,\ j_m\} \bullet \lambda\ x : \mathbb{W} \upharpoonright n \bullet u\ x\ y\} \end{array}$$

Given a model M and generic argument sets w_1, ..., w_n, the semantic value of the schema e in that model overridden by the association of the generic parameter names with those sets is w. All combinations of generic argument sets are considered. The function u maps the generic argument sets to a binding in the schema w. The paragraph relates the model M to that model extended with the binding that associates the names of the schema e (namely j_1, ..., j_m) with the corresponding value in the binding resulting from application of u to arbitrary instantiating sets x. Associations for names decorated with the reserved stroke ♠ are also introduced whilst determining the semantic value of e, so that references to them from generic types (see 15.2.6.2) can avoid being captured.

C.4.6 Generic schema definition

C.4.6.1 Syntax

```
Paragraph       = ...
                | GENSCH , NAME , [-tok , Formals , ]-tok , SchemaText , END
                | ...
                ;
```


C.4.6.2 Transformation

The generic schema definition paragraph GENSCH i $[i_1, ..., i_n]$ t END can be instantiated to produce a schema definition paragraph.

$$\texttt{GENSCH}\ i\ [i_1, ..., i_n]\ t\ \texttt{END} \implies \texttt{GENAX}\ [i_1, ..., i_n]\ [i == t]\ \texttt{END}$$

It is semantically equivalent to the generic axiomatic description paragraph with the same generic parameters and whose sole declaration associates the schema's name with the expression resulting from syntactic transformation of the schema text.

C.4.7 Horizontal definition

C.4.7.1 Syntax

```
Paragraph       = ...
                | ZED , DeclName , == , Expression , END
                | ...
                ;
```

C.4.7.2 Transformation

The horizontal definition paragraph ZED $i == e$ END introduces the global name i, associating with it the value of e.

$$\texttt{ZED}\ i == e\ \texttt{END} \implies \texttt{AX}\ [i == e]\ \texttt{END}$$

It is semantically equivalent to the axiomatic description paragraph that introduces the same single declaration.

C.4.8 Generic horizontal definition

C.4.8.1 Syntax

```
Paragraph       = ...
                | ZED , NAME , [-tok , Formals , ]-tok , == , Expression , END
                | ...
                ;
```

C.4.8.2 Transformation

The generic horizontal definition paragraph ZED i $[i_1, ..., i_n] == e$ END can be instantiated to produce a horizontal definition paragraph.

$$\texttt{ZED}\ i\ [i_1, ..., i_n] == e\ \texttt{END} \implies \texttt{GENAX}\ [i_1, ..., i_n]\ [i == e]\ \texttt{END}$$

It is semantically equivalent to the generic axiomatic description paragraph with the same generic parameters and that introduces the same single declaration.

C.4.9 Generic operator definition

C.4.9.1 Syntax

```
Paragraph        = ...
                 | ZED , GenName , == , Expression , END
                 | ...
                 ;

GenName          = PrefixGenName
                 | PostfixGenName
                 | InfixGenName
                 | NofixGenName
                 ;

PrefixGenName    = PRE , NAME
                 | L , { NAME , ( ES | SS ) } , NAME , ( ERE | SRE ) , NAME
                 ;

PostfixGenName   = NAME , POST
                 | NAME , EL , { NAME , ( ES | SS ) } , NAME , ( ER | SR )
                 ;

InfixGenName     = NAME , I , NAME
                 | NAME , EL , { NAME , ( ES | SS ) } , NAME , ( ERE | SRE ) , NAME
                 ;

NofixGenName     = L , { NAME , ( ES | SS ) } , NAME , ( ER | SR ) ;
```

C.4.9.2 Transformation

All generic names are transformed to juxtapositions of NAMEs and generic parameter lists. This causes the generic operator definition paragraphs in which they appear to become generic horizontal definition paragraphs, and thus be amenable to further syntactic transformation.

Each resulting NAME shall be one for which there is an operator template paragraph in scope (see 12.2.8).

C.4.9.3 PrefixGenName

$$
\begin{aligned}
pre\ i &\Longrightarrow pre{\bowtie}\,[i] \\
ln\ i_1\ ess_1\ \ldots\ i_{n-2}\ ess_{n-2}\ i_{n-1}\ ere\ i_n &\Longrightarrow ln{\bowtie}ess_1 \ldots {\bowtie}ess_{n-2}{\bowtie}ere{\bowtie}\,[i_1, \ldots, i_{n-2}, i_{n-1}, i_n] \\
ln\ i_1\ ess_1\ \ldots\ i_{n-2}\ ess_{n-2}\ i_{n-1}\ sre\ i_n &\Longrightarrow ln{\bowtie}ess_1 \ldots {\bowtie}ess_{n-2}{\bowtie}sre{\bowtie}\,[i_1, \ldots, i_{n-2}, i_{n-1}, i_n]
\end{aligned}
$$

C.4.9.4 PostfixGenName

$$
\begin{aligned}
i\ post &\Longrightarrow {\bowtie}post\ [i] \\
i_1 el\ i_2\ ess_2\ \ldots\ i_{n-1}\ ess_{n-1}\ i_n\ er &\Longrightarrow {\bowtie}el{\bowtie}ess_2 \ldots {\bowtie}ess_{n-1}{\bowtie}er\ [i_1, i_2, \ldots, i_{n-1}, i_n] \\
i_1 el\ i_2\ ess_2\ \ldots\ i_{n-1}\ ess_{n-1}\ i_n\ sr &\Longrightarrow {\bowtie}el{\bowtie}ess_2 \ldots {\bowtie}ess_{n-1}{\bowtie}sr\ [i_1, i_2, \ldots, i_{n-1}, i_n]
\end{aligned}
$$

C.4.9.5 InfixGenName

$$
\begin{aligned}
i_1 in\ i_2 &\implies \bowtie in \bowtie\ [i_1, i_2] \\
i_1 el\ i_2\ ess_2\ ...\ i_{n-2}\ ess_{n-2}\ i_{n-1}\ ere\ i_n &\implies \bowtie el \bowtie ess_2 ... \bowtie ess_{n-2} \bowtie ere \bowtie\ [i_1, i_2, ..., i_{n-2}, i_{n-1}, i_n] \\
i_1 el\ i_2\ ess_2\ ...\ i_{n-2}\ ess_{n-2}\ i_{n-1}\ sre\ i_n &\implies \bowtie el \bowtie ess_2 ... \bowtie ess_{n-2} \bowtie sre \bowtie\ [i_1, i_2, ..., i_{n-2}, i_{n-1}, i_n]
\end{aligned}
$$

C.4.9.6 NofixGenName

$$
\begin{aligned}
ln\ i_1\ ess_1\ ...\ i_{n-1}\ ess_{n-1}\ i_n\ er &\implies ln \bowtie ess_1 ... \bowtie ess_{n-1} \bowtie er\ [i_1, ..., i_{n-1}, i_n] \\
ln\ i_1\ ess_1\ ...\ i_{n-1}\ ess_{n-1}\ i_n\ sr &\implies ln \bowtie ess_1 ... \bowtie ess_{n-1} \bowtie sr\ [i_1, ..., i_{n-1}, i_n]
\end{aligned}
$$

C.4.10 Free types

C.4.10.1 Syntax

```
Paragraph       = ...
                | ZED , Freetype , { & , Freetype } , END
                | ...
                ;

Freetype        = NAME , ::= , Branch , { |-tok , Branch } ;        (* free type *)

Branch          = DeclName , [ ⟪ , Expression , ⟫ ] ;               (* element or injection *)
```

C.4.10.2 Transformation

The transformation of free types paragraphs is done in two stages. First, the branches are permuted to bring elements to the front and injections to the rear.

$$
... \mid g\langle\langle e \rangle\rangle \mid h \mid ... \implies ... \mid h \mid g\langle\langle e \rangle\rangle \mid ...
$$

Exhaustive application of this syntactic transformation rule effects a sort.

The second stage requires implicit generic instantiations to have been filled in, which is done during typechecking (see 13.2.3.3). Hence that second stage is delayed until after typechecking, where it appears in the form of a semantic transformation rule (see 14.2.3.1).

C.4.10.3 Type

$$\frac{\begin{array}{c}\beta \vdash^{\mathcal{E}} e_{1\,1} \mathbin{⦂} \tau_{1\,1} \quad \ldots \quad \beta \vdash^{\mathcal{E}} e_{1\,n_1} \mathbin{⦂} \tau_{1\,n_1} \\ \vdots \\ \beta \vdash^{\mathcal{E}} e_{r\,1} \mathbin{⦂} \tau_{r\,1} \quad \ldots \quad \beta \vdash^{\mathcal{E}} e_{r\,n_r} \mathbin{⦂} \tau_{r\,n_r}\end{array}}{\begin{array}{l}\Sigma \vdash^{\mathcal{D}} \texttt{ZED} \\ \quad f_1 ::= h_{1\,1} \mid \ldots \mid h_{1\,m_1} \mid \\ \qquad g_{1\,1}\langle\!\langle e_{1\,1} \mathbin{⦂} \tau_{1\,1}\rangle\!\rangle \mid \\ \qquad \vdots \mid \\ \qquad g_{1\,n_1}\langle\!\langle e_{1\,n_1} \mathbin{⦂} \tau_{1\,n_1}\rangle\!\rangle \;\&\; \\ \quad \vdots \;\& \\ \quad f_r ::= h_{r\,1} \mid \ldots \mid h_{r\,m_r} \mid \\ \qquad g_{r\,1}\langle\!\langle e_{r\,1} \mathbin{⦂} \tau_{r\,1}\rangle\!\rangle \mid \\ \qquad \vdots \mid \\ \qquad g_{r\,n_r}\langle\!\langle e_{r\,n_r} \mathbin{⦂} \tau_{r\,n_r}\rangle\!\rangle \\ \texttt{END} \mathbin{⦂} \sigma\end{array}} \Rightarrow \left(\begin{array}{l}\#\{f_1,\ h_{1\,1},\ \ldots,\ h_{1\,m_1},\ g_{1\,1},\ \ldots,\ g_{1\,n_1}, \\ \vdots, \\ \qquad f_r,\ h_{r\,1},\ \ldots,\ h_{r\,m_r},\ g_{r\,1},\ \ldots,\ g_{r\,n_r}\} \\ \quad = r + m_1 + \ldots + m_r + n_1 + \ldots + n_r \\ \beta = \Sigma \oplus \{f_1 \mapsto \mathbb{P}(\texttt{GIVEN}\ f_1),\ \ldots,\ f_r \mapsto \mathbb{P}(\texttt{GIVEN}\ f_r)\} \\ \tau_{1\,1} = \mathbb{P}\,\alpha_{1\,1} \quad \ldots \quad \tau_{1\,n_1} = \mathbb{P}\,\alpha_{1\,n_1} \\ \qquad \vdots \\ \tau_{r\,1} = \mathbb{P}\,\alpha_{r\,1} \quad \ldots \quad \tau_{r\,n_r} = \mathbb{P}\,\alpha_{r\,n_r} \\ \sigma \;=\; f_1 : \mathbb{P}(\texttt{GIVEN}\ f_1); \\ \qquad\qquad h_{1\,1} : \texttt{GIVEN}\ f_1;\ \ldots;\ h_{1\,m_1} : \texttt{GIVEN}\ f_1; \\ \qquad\qquad g_{1\,1} : \mathbb{P}(\tau_{1\,1} \times \texttt{GIVEN}\ f_1); \\ \qquad\qquad\quad \vdots; \\ \qquad\qquad g_{1\,n_1} : \mathbb{P}(\tau_{1\,n_1} \times \texttt{GIVEN}\ f_1); \\ \qquad\quad \vdots; \\ \qquad f_r : \mathbb{P}(\texttt{GIVEN}\ f_r); \\ \qquad\qquad h_{r\,1} : \texttt{GIVEN}\ f_r;\ \ldots;\ h_{r\,m_r} : \texttt{GIVEN}\ f_r; \\ \qquad\qquad g_{r\,1} : \mathbb{P}(\tau_{r\,1} \times \texttt{GIVEN}\ f_r); \\ \qquad\qquad\quad \vdots; \\ \qquad\qquad g_{r\,n_r} : \mathbb{P}(\tau_{r\,n_r} \times \texttt{GIVEN}\ f_r)\end{array}\right)$$

In a free types paragraph, the names of the free types, elements and injections shall all be different. The expressions representing the domains of the injections are typechecked in an environment overridden by the names of the free types, and shall all be sets. The annotation of the paragraph is the signature whose names are those of all the free types, the elements, and the injections, each associated with the corresponding type.

C.4.10.4 Semantics

A free types paragraph is semantically equivalent to the sequence of given type paragraph and axiomatic definition paragraph defined here.

NOTE 1 This exploits notation that is not present in the annotated syntax for the purpose of abbreviation.

$$\begin{array}{l}\texttt{ZED} \\ f_1 ::= h_{1\,1} \mid \ldots \mid h_{1\,m_1} \mid g_{1\,1}\langle\!\langle e_{1\,1}\rangle\!\rangle \mid \ldots \mid g_{1\,n_1}\langle\!\langle e_{1\,n_1}\rangle\!\rangle \\ \;\& \ \ldots\ \& \\ f_r ::= h_{r\,1} \mid \ldots \mid h_{r\,m_r} \mid g_{r\,1}\langle\!\langle e_{r\,1}\rangle\!\rangle \mid \ldots \mid g_{r\,n_r}\langle\!\langle e_{r\,n_r}\rangle\!\rangle \\ \texttt{END}\end{array}$$

$$\Longrightarrow$$

$$\begin{array}{l}\texttt{ZED} \\ [f_1, \ldots, f_r] \\ \texttt{END}\end{array}$$

$$
\begin{array}{l}
\texttt{AX} \\
h_{1\,1},\ ...,\ h_{1\,m_1} : f_1 \\
\vdots \\
h_{r\,1},\ ...,\ h_{r\,m_r} : f_r \\
g_{1\,1} : \mathbb{P}(e_{1\,1} \times f_1);\ ...;\ g_{1\,n_1} : \mathbb{P}(e_{1\,n_1} \times f_1) \\
\vdots \\
g_{r\,1} : \mathbb{P}(e_{r\,1} \times f_r);\ ...;\ g_{r\,n_r} : \mathbb{P}(e_{r\,n_r} \times f_r) \\
| \\
(\forall\ u : e_{1\,1} \bullet \exists_1\ x : g_{1\,1} \bullet x\ .\ 1 = u) \land ... \land (\forall\ u : e_{1\,n_1} \bullet \exists_1\ x : g_{1\,n_1} \bullet x\ .\ 1 = u) \\
\vdots \land \\
(\forall\ u : e_{r\,1} \bullet \exists_1\ x : g_{r\,1} \bullet x\ .\ 1 = u) \land ... \land (\forall\ u : e_{r\,n_r} \bullet \exists_1\ x : g_{r\,n_r} \bullet x\ .\ 1 = u) \\
\\
(\forall\ u, v : e_{1\,1} \mid g_{1\,1} u = g_{1\,1} v \bullet u = v) \land ... \land (\forall\ u, v : e_{1\,n_1} \mid g_{1\,n_1} u = g_{1\,n_1} v \bullet u = v) \\
\vdots \land \\
(\forall\ u, v : e_{r\,1} \mid g_{r\,1} u = g_{r\,1} v \bullet u = v) \land ... \land (\forall\ u, v : e_{r\,n_r} \mid g_{r\,n_r} u = g_{r\,n_r} v \bullet u = v) \\
\\
\forall\ b_1, b_2 : \mathbb{N} \bullet \\
\quad (\forall\ w : f_1 \mid \\
\qquad (b_1 = 1 \land w = h_{1\,1} \lor ... \lor b_1 = m_1 \land w = h_{1\,m_1} \lor \\
\qquad\quad b_1 = m_1 + 1 \land w \in \{x : g_{1\,1} \bullet x\ .\ 2\} \lor ... \lor b_1 = m_1 + n_1 \land w \in \{x : g_{1\,n_1} \bullet x\ .\ 2\}) \\
\qquad \land (b_2 = 1 \land w = h_{1\,1} \lor ... \lor b_2 = m_1 \land w = h_{1\,m_1} \lor \\
\qquad\quad b_2 = m_1 + 1 \land w \in \{x : g_{1\,1} \bullet x\ .\ 2\} \lor ... \lor b_2 = m_1 + n_1 \land w \in \{x : g_{1\,n_1} \bullet x\ .\ 2\}) \bullet \\
\qquad\quad b_1 = b_2) \land \\
\quad \vdots \land \\
\quad (\forall\ w : f_r \mid \\
\qquad (b_1 = 1 \land w = h_{r\,1} \lor ... \lor b_1 = m_r \land w = h_{r\,m_r} \lor \\
\qquad\quad b_1 = m_r + 1 \land w \in \{x : g_{r\,1} \bullet x\ .\ 2\} \lor ... \lor b_1 = m_r + n_r \land w \in \{x : g_{r\,n_r} \bullet x\ .\ 2\}) \\
\qquad \land (b_2 = 1 \land w = h_{r\,1} \lor ... \lor b_2 = m_r \land w = h_{r\,m_r} \lor \\
\qquad\quad b_2 = m_r + 1 \land w \in \{x : g_{r\,1} \bullet x\ .\ 2\} \lor ... \lor b_2 = m_r + n_r \land w \in \{x : g_{r\,n_r} \bullet x\ .\ 2\}) \bullet \\
\qquad\quad b_1 = b_2) \\
\\
\forall\ w_1 : \mathbb{P}\, f_1;\ ...;\ w_r : \mathbb{P}\, f_r \mid \\
\qquad h_{1\,1} \in w_1 \land ... \land h_{1\,m_1} \in w_1 \land \\
\qquad \vdots \land \\
\qquad h_{r\,1} \in w_r \land ... \land h_{r\,m_r} \in w_r \land \\
\qquad (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{1\,1}) \bullet g_{1\,1} y \in w_1) \land \\
\qquad ... \land (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{1\,n_1}) \bullet g_{1\,n_1} y \in w_1) \land \\
\qquad \vdots \land \\
\qquad (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{r\,1}) \bullet g_{r\,1} y \in w_r) \land \\
\qquad ... \land (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{r\,n_r}) \bullet g_{r\,n_r} y \in w_r) \bullet \\
\quad w_1 = f_1 \land ... \land w_r = f_r \\
\texttt{END}
\end{array}
$$

The type names are introduced by the given types paragraph. The elements are declared as members of their corresponding free types. The injections are declared as functions from values in their domains to their corresponding free type.

The first of the four blank-line separated predicates is the total functionality property. It ensures that for every injection, the injection is functional at every value in its domain.

The second of the four blank-line separated predicates is the injectivity property. It ensures that for every injection, any pair of values in its domain for which the injection returns the same value shall be a pair of equal values (hence the name injection).

The third of the four blank-line separated predicates is the disjointness property. It ensures that for every free type, every pair of values of the free type are equal only if they are the same element or are returned by application of the same injection to equal values.

The fourth of the four blank-line separated predicates is the induction property. It ensures that for every free type, its members are its elements, the values returned by its injections, and nothing else.

The generated μ expressions in the induction property are intended to effect substitutions of all references to the free type names, including any such references within generic instantiation lists in the e expressions.

NOTE 2 That is why this is a semantic transformation not a syntactic one: all implicit generic instantiations shall have been made explicit before it is applied.

NOTE 3 The right-hand side of this transformation could have been expressed using notation from the mathematical toolkit, as follows, but for the desire to define the Z core language separately from the mathematical toolkit.

ZED
$[f_1, ..., f_r]$
END

AX
$h_{1\,1}, \ ..., \ h_{1\,m_1} : f_1$
$\vdots$
$h_{r\,1}, \ ..., \ h_{r\,m_r} : f_r$
$g_{1\,1} : e_{1\,1} \rightarrowtail f_1; \ ...; \ g_{1\,n_1} : e_{1\,n_1} \rightarrowtail f_1$
$\vdots$
$g_{r\,1} : e_{r\,1} \rightarrowtail f_r; \ ...; \ g_{r\,n_r} : e_{r\,n_r} \rightarrowtail f_r$
$|$
$disjoint\langle \{h_{1\,1}\}, \ ..., \ \{h_{1\,m_1}\}, \ ran\, g_{1\,1}, \ ..., \ ran\, g_{1\,n_1} \rangle$
$\vdots$
$disjoint\langle \{h_{r\,1}\}, \ ..., \ \{h_{r\,m_r}\}, \ ran\, g_{r\,1}, \ ..., \ ran\, g_{r\,n_r} \rangle$
$\forall \ w_1 : \mathbb{P}\, f_1; \ ...; \ w_r : \mathbb{P}\, f_r \mid$
$\{h_{1\,1}, \ ..., \ h_{1\,m_1}\} \cup g_{1\,1}(\!| \ \mu\, f_1 == w_1; \ ...; \ f_r == w_r \bullet e_{1\,1} \ |\!)$
$\cup ... \cup g_{1\,n_1}(\!| \ \mu\, f_1 == w_1; \ ...; \ f_r == w_r \bullet e_{1\,n_1} \ |\!) \subseteq w_1 \ \land$
$\vdots \ \land$
$\{h_{r\,1}, \ ..., \ h_{r\,m_r}\} \cup g_{r\,1}(\!| \ \mu\, f_1 == w_1; \ ...; \ f_r == w_r \bullet e_{r\,1} \ |\!)$
$\cup ... \cup g_{r\,n_r}(\!| \ \mu\, f_1 == w_1; \ ...; \ f_r == w_r \bullet e_{r\,n_r} \ |\!) \subseteq w_r \bullet$
$w_1 = f_1 \land ... \land w_r = f_r$
END

C.4.11 Conjecture

C.4.11.1 Syntax

```
Paragraph        = ...
                 | ZED , ⊢? , Predicate , END
                 | ...
                 ;
```

C.4.11.2 Type

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \texttt{ZED} \vdash ?\ p\ \texttt{END} \ \wr\ \sigma}(\sigma = \epsilon)$$

In a conjecture paragraph $\texttt{ZED} \vdash ?\ p\ \texttt{END}$, the predicate p shall be well-typed. The annotation of the paragraph is the empty signature.

C.4.11.3 Semantics

The conjecture paragraph $\texttt{ZED} \vdash ?\ p\ \texttt{END}$ expresses a property that may logically follow from the specification. It may be a starting point for a proof.

$$[\![\texttt{ZED} \vdash ?\ p\ \texttt{END}]\!]^{\mathcal{D}} = id\ Model$$

It relates a model to itself: the truth of p in a model does not affect the meaning of the specification.

C.4.12 Generic conjecture

C.4.12.1 Syntax

```
Paragraph        = ...
                 | ZED , [-tok , Formals , ]-tok , ⊢? , Predicate , END
                 | ...
                 ;
```

C.4.12.2 Type

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_1), \ldots, i_n \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_n)\} \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \texttt{ZED}\ [i_1, \ldots, i_n] \vdash ?\ p\ \texttt{END} \ \wr\ \sigma}\begin{pmatrix} \#\{i_1, \ldots, i_n\} = n \\ \sigma = \epsilon \end{pmatrix}$$

In a generic conjecture paragraph $\texttt{ZED}\ [i_1, \ldots, i_n] \vdash ?\ p\ \texttt{END}$, there shall be no duplication of names within the generic parameters. The predicate p shall be well-typed in an environment overridden by the generic parameters. The annotation of the paragraph is the empty signature.

C.4.12.3 Semantics

The generic conjecture paragraph $\texttt{ZED}\ [i_1, \ldots, i_n] \vdash ?\ p\ \texttt{END}$ expresses a generic property that may logically follow from the specification. It may be a starting point for a proof.

$$[\![\texttt{ZED}\ [i_1, \ldots, i_n] \vdash ?\ p\ \texttt{END}]\!]^{\mathcal{D}} = id\ Model$$

It relates a model to itself: the truth of p in a model does not affect the meaning of the specification.

C.4.13 Operator template

An operator template has only syntactic significance: it notifies the reader to treat all occurrences in this section of the words in the template, with whatever strokes they are decorated, as particular prefix, infix, postfix or nofix names. The category of the operator—relation, function, or generic—determines how applications of the operator are interpreted— as relational predicates, application expressions, or generic instantiation expressions respectively.

C.4.13.1 Syntax

```
Paragraph          = ...
                   | ZED , OperatorTemplate , END
                   ;

OperatorTemplate   = relation , Template
                   | function , CategoryTemplate
                   | generic , CategoryTemplate
                   ;

CategoryTemplate   = PrefixTemplate
                   | PostfixTemplate
                   | Prec , Assoc , InfixTemplate
                   | NofixTemplate
                   ;

Prec               = NUMERAL ;

Assoc              = leftassoc
                   | rightassoc
                   ;

Template           = PrefixTemplate
                   | PostfixTemplate
                   | InfixTemplate
                   | NofixTemplate
                   ;

PrefixTemplate     = (-tok , PrefixName , )-tok
                   | (-tok , ℙ , _ , )-tok
                   ;

PostfixTemplate    = (-tok , PostfixName , )-tok ;

InfixTemplate      = (-tok , InfixName , )-tok ;

NofixTemplate      = (-tok , NofixName , )-tok ;
```

C.5 Predicate

C.5.1 Introduction

A `Predicate` expresses constraints between the values associated with names. A `Predicate` can be any of universal quantification, existential quantification, unique existential quantification, newline conjunction, semicolon conjunction, equivalence, implication, disjunction, conjunction, negation, relation operator application, membership, schema predicate, truth, falsity, or parenthesized predicate.

C.5.2 Universal quantification

C.5.2.1 Syntax

```
Predicate        = ∀ , SchemaText , • , Predicate
                 | ...
                 ;
```

C.5.2.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \ ⦂ \ \tau \qquad \Sigma \oplus \beta \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \forall (e \ ⦂ \ \tau) \bullet p}(\tau = \mathbb{P}[\beta])$$

In a universal quantification predicate $\forall\ e \bullet p$, expression e shall be a schema, and predicate p shall be well-typed in the environment overridden by the signature of schema e.

C.5.2.3 Semantics

The universal quantification predicate $\forall\ e \bullet p$ is *true* if and only if predicate p is *true* for all bindings of the schema e.

$$[\![\ \forall\ e \bullet p\ ]\!]^{\mathcal{P}} \quad = \quad \{M : Model \mid \forall\ t : [\![\ e\ ]\!]^{\mathcal{E}} M \bullet M \oplus t \in [\![\ p\ ]\!]^{\mathcal{P}} \bullet M\}$$

In terms of the semantic universe, it is *true* in those models for which p is *true* in that model overridden by all bindings in the semantic value of e, and is *false* otherwise.

C.5.3 Existential quantification

C.5.3.1 Syntax

```
Predicate        = ...
                 | ∃ , SchemaText , • , Predicate
                 | ...
                 ;
```

C.5.3.2 Transformation

The existential quantification predicate $\exists\ t \bullet p$ is *true* if and only if p is *true* for at least one value of t.

$$\exists\ t \bullet p \quad \Longrightarrow \quad \neg\ \forall\ t \bullet \neg\ p$$

It is semantically equivalent to p being *false* for not all values of t.

C.5.4 Unique existential quantification

C.5.4.1 Syntax

```
Predicate        = ...
                 | ∃1 , SchemaText , • , Predicate
                 | ...
                 ;
```

C.5.4.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \ ⦂ \ \tau \qquad \Sigma \oplus \beta \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \exists_1 (e \ ⦂ \ \tau) \bullet p}(\tau = \mathbb{P}[\beta])$$

In a unique existential quantification predicate $\exists_1\ e \bullet p$, expression e shall be a schema, and predicate p shall be well-typed in the environment overridden by the signature of schema e.

C.5.4.3 Semantics

The unique existential quantification predicate $\exists_1\ e \bullet p$ is *true* if and only if there is exactly one value for e for which p is *true*.

$$\exists_1\ e \bullet p \quad \Longrightarrow \quad \neg\ (\forall\ e \bullet \neg\ (p \land (\forall\ [e \mid p]^{\bowtie} \bullet \theta\ e = \theta\ e\ ^{\bowtie})))$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$\exists_1\ e \bullet p \quad \Longrightarrow \quad \exists\ e \bullet p \land (\forall\ [e \mid p]^{\bowtie} \bullet \theta\ e = \theta\ e\ ^{\bowtie})$$

It is semantically equivalent to there existing at least one value for e for which p is *true* and all those values for which it is *true* being the same.

C.5.5 Newline conjunction

C.5.5.1 Syntax

```
Predicate        = ...
                 | Predicate , NL , Predicate
                 | ...
                 ;
```

C.5.5.2 Transformation

The newline conjunction predicate p_1 NL p_2 is *true* if and only if both its predicates are *true*.

$$p_1\ \texttt{NL}\ p_2 \quad \Longrightarrow \quad p_1 \land p_2$$

It is semantically equivalent to the conjunction predicate $p_1 \land p_2$.

C.5.6 Semicolon conjunction

C.5.6.1 Syntax

```
Predicate        = ...
                 | Predicate , ;-tok , Predicate
                 | ...
                 ;
```

C.5.6.2 Transformation

The semicolon conjunction predicate $p_1;\ p_2$ is *true* if and only if both its predicates are *true*.

$$p_1;\ p_2 \quad \Longrightarrow \quad p_1 \land p_2$$

It is semantically equivalent to the conjunction predicate $p_1 \land p_2$.

C.5.7 Equivalence

C.5.7.1 Syntax

```
Predicate       = ...
                | Predicate , ⇔ , Predicate
                | ...
                ;
```

C.5.7.2 Transformation

The equivalence predicate $p_1 \Leftrightarrow p_2$ is *true* if and only if both p_1 and p_2 are *true* or neither is *true*.

$$p_1 \Leftrightarrow p_2 \quad \Longrightarrow \quad (p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$$

It is semantically equivalent to each of p_1 and p_2 being *true* if the other is *true*.

C.5.8 Implication

C.5.8.1 Syntax

```
Predicate       = ...
                | Predicate , ⇒ , Predicate
                | ...
                ;
```

C.5.8.2 Transformation

The implication predicate $p_1 \Rightarrow p_2$ is *true* if and only if p_2 is *true* if p_1 is *true*.

$$p_1 \Rightarrow p_2 \quad \Longrightarrow \quad \neg\, p_1 \vee p_2$$

It is semantically equivalent to p_1 being *false* disjoined with p_2 being *true*.

C.5.9 Disjunction

C.5.9.1 Syntax

```
Predicate       = ...
                | Predicate , ∨ , Predicate
                | ...
                ;
```

C.5.9.2 Transformation

The disjunction predicate $p_1 \vee p_2$ is *true* if and only if at least one of p_1 and p_2 is *true*.

$$p_1 \vee p_2 \quad \Longrightarrow \quad \neg\, (\neg\, p_1 \wedge \neg\, p_2)$$

It is semantically equivalent to not both of p_1 and p_2 being *false*.

C.5.10 Conjunction

C.5.10.1 Syntax

```
Predicate       = ...
                | Predicate , ∧ , Predicate
                | ...
                ;
```

C.5.10.2 Type

$$\frac{\Sigma \vdash^{\mathcal{P}} p_1 \qquad \Sigma \vdash^{\mathcal{P}} p_2}{\Sigma \vdash^{\mathcal{P}} p_1 \wedge p_2}$$

A conjunction predicate $p_1 \wedge p_2$ is well-typed if and only if predicates p_1 and p_2 are well-typed.

C.5.10.3 Semantics

The conjunction predicate $p_1 \wedge p_2$ is *true* if and only if p_1 and p_2 are *true*.

$$[\![p_1 \wedge p_2]\!]^{\mathcal{P}} = [\![p_1]\!]^{\mathcal{P}} \cap [\![p_2]\!]^{\mathcal{P}}$$

In terms of the semantic universe, it is *true* in those models in which both p_1 and p_2 are *true*, and is *false* otherwise.

C.5.11 Negation

C.5.11.1 Syntax

```
Predicate       = ...
                | ¬ , Predicate
                | ...
                ;
```

C.5.11.2 Type

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \neg p}$$

A negation predicate $\neg p$ is well-typed if and only if predicate p is well-typed.

C.5.11.3 Semantics

The negation predicate $\neg p$ is *true* if and only if p is *false*.

$$[\![\neg p]\!]^{\mathcal{P}} = \mathit{Model} \setminus [\![p]\!]^{\mathcal{P}}$$

In terms of the semantic universe, it is *true* in all models except those in which p is *true*.

C.5.12 Relation operator application

C.5.12.1 Syntax

```
Predicate       = ...
                | Relation
                | ...
                ;
```



```
Relation          = PrefixRel
                  | PostfixRel
                  | InfixRel
                  | NofixRel
                  ;

PrefixRel         = PREP , Expression
                  | LP , ExpSep , ( Expression , EREP | ExpressionList , SREP ) , Expression
                  ;

PostfixRel        = Expression , POSTP
                  | Expression , ELP , ExpSep , ( Expression , ERP | ExpressionList , SRP )
                  ;

InfixRel          = Expression , ( ∈ | =-tok | IP ) , Expression ,
                      { ( ∈ | =-tok | IP ) , Expression }
                  | Expression , ELP , ExpSep ,
                    ( Expression , EREP | ExpressionList , SREP ) , Expression
                  ;

NofixRel          = LP , ExpSep , ( Expression , ERP | ExpressionList , SRP ) ;
```

C.5.12.2 Transformation

All relation operator applications are transformed to annotated membership predicates.

Each relation `NAME` shall be one for which there is an operator template paragraph in scope (see 12.2.8).

The left-hand sides of many of these transformation rules involve `ExpSep` phrases: they use *es* metavariables. None of them use *ss* metavariables: in other words, only the `Expression ES` case of `ExpSep` is specified, not the `ExpressionList SS` case. Where the latter case occurs in a specification, the `ExpressionList` shall be transformed by the rule in 12.2.12 to an expression, and thence a transformation analogous to that specified for the former case can be performed, differing only in that a *ss* appears in the relation name in place of an *es*.

C.5.12.2.1 `PrefixRel`

$$
\begin{aligned}
prep\ e &\implies e \in prep\bowtie \\
lp\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ erep\ e_n &\implies (e_1, \ldots, e_{n-2}, e_{n-1}, e_n) \in lp\bowtie es_1 \ldots \bowtie es_{n-2}\bowtie erep\bowtie \\
lp\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ srep\ e_n &\implies (e_1, \ldots, e_{n-2}, a_{n-1}, e_n) \in lp\bowtie es_1 \ldots \bowtie es_{n-2}\bowtie srep\bowtie
\end{aligned}
$$

C.5.12.2.2 `PostfixRel`

$$
\begin{aligned}
e\ postp &\implies e \in \bowtie postp \\
e_1\ elp\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ e_n\ erp &\implies (e_1, e_2, \ldots, e_{n-1}, e_n) \in \bowtie elp\bowtie es_2 \ldots \bowtie es_{n-1}\bowtie erp \\
e_1\ elp\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ a_n\ srp &\implies (e_1, e_2, \ldots, e_{n-1}, a_n) \in \bowtie elp\bowtie es_2 \ldots \bowtie es_{n-1}\bowtie srp
\end{aligned}
$$

C.5.12.2.3 InfixRel

$$e_1\ ip_1\ e_2\ ip_2\ e_3\ ... \quad \Longrightarrow \quad e_1\ ip_1\ (e_2 \mathbin{⦂} \alpha_1) \land (e_2 \mathbin{⦂} \alpha_1)\ ip_2\ (e_3 \mathbin{⦂} \alpha_2)\ ...$$

The chained relation $e_1\ ip_1\ e_2\ ip_2\ e_3\ ...$ is semantically equivalent to a conjunction of relational predicates, with the constraint that duplicated expressions be of the same type.

$$\begin{array}{lcl} e_1 = e_2 & \Longrightarrow & e_1 \in \{e_2\} \\ e_1\ ip\ e_2 & \Longrightarrow & (e_1, e_2) \in \bowtie ip \bowtie \end{array}$$

ip in the above transformation is excluded from being $\in$ or $=$, whereas $ip_1, ip_2, ...$ in the chained relation can be $\in$ or $=$.

$$\begin{array}{lcl} e_1\ elp\ e_2\ es_2\ ...\ e_{n-2}\ es_{n-2}\ e_{n-1}\ erep\ e_n & \Longrightarrow & (e_1, e_2, ..., e_{n-2}, e_{n-1}, e_n) \in \bowtie elp \bowtie es_2 ... \bowtie es_{n-2} \bowtie erep \bowtie \\ e_1\ elp\ e_2\ es_2\ ...\ e_{n-2}\ es_{n-2}\ a_{n-1}\ srep\ e_n & \Longrightarrow & (e_1, e_2, ..., e_{n-2}, a_{n-1}, e_n) \in \bowtie elp \bowtie es_2 ... \bowtie es_{n-2} \bowtie srep \bowtie \end{array}$$

C.5.12.2.4 NofixRel

$$\begin{array}{lcl} lp\ e_1\ es_1\ ...\ e_{n-1}\ es_{n-1}\ e_n\ erp & \Longrightarrow & (e_1, ..., e_{n-1}, e_n) \in lp \bowtie es_1 ... \bowtie es_{n-1} \bowtie erp \\ lp\ e_1\ es_1\ ...\ e_{n-1}\ es_{n-1}\ a_n\ srp & \Longrightarrow & (e_1, ..., e_{n-1}, a_n) \in lp \bowtie es_1 ... \bowtie es_{n-1} \bowtie srp \end{array}$$

C.5.12.3 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{⦂} \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathbin{⦂} \tau_2}{\Sigma \vdash^{\mathcal{P}} (e_1 \mathbin{⦂} \tau_1) \in (e_2 \mathbin{⦂} \tau_2)} \left(\tau_2 = \mathbb{P}\, \tau_1 \right)$$

In a membership predicate $e_1 \in e_2$, expression e_2 shall be a set, and expression e_1 shall be of the same type as the members of set e_2.

C.5.12.4 Semantics

The membership predicate $e_1 \in e_2$ is *true* if and only if the value of e_1 is in the set that is the value of e_2.

$$[\![\ e_1 \in e_2\ ]\!]^{\mathcal{P}} \quad = \quad \{M : Model \mid [\![\ e_1\ ]\!]^{\mathcal{E}} M \in [\![\ e_2\ ]\!]^{\mathcal{E}} M \bullet M\}$$

In terms of the semantic universe, it is *true* in those models in which the semantic value of e_1 is in the semantic value of e_2, and is *false* otherwise.

C.5.13 Schema

C.5.13.1 Syntax

```
Predicate       = ...
                | Expression
                | ...
                ;
```


C.5.13.2 Transformation

The schema predicate e is *true* if and only if the binding of the names in the signature of schema e satisfies the constraints of that schema.

$$e \implies \theta\, e \in e$$

It is semantically equivalent to the binding constructed by $\theta\, e$ being a member of the set denoted by schema e.

C.5.14 Truth

C.5.14.1 Syntax

```
Predicate       = ...
                | true
                | ...
                ;
```

C.5.14.2 Type

$$\frac{}{\Sigma \vdash^{\mathcal{P}} \text{true}}$$

A truth predicate is always well-typed.

C.5.14.3 Semantics

A truth predicate is always *true*.

$$[\![\text{ true }]\!]^{\mathcal{P}} = \mathit{Model}$$

In terms of the semantic universe, it is *true* in all models.

C.5.15 Falsity

C.5.15.1 Syntax

```
Predicate       = ...
                | false
                | ...
                ;
```

C.5.15.2 Transformation

The falsity predicate false is never *true*.

$$\text{false} \implies \neg\ \text{true}$$

It is semantically equivalent to the negation of true.

C.5.16 Parenthesized predicate

C.5.16.1 Syntax

```
Predicate       = ...
                | (-tok , Predicate , )-tok
                ;
```

C.5.16.2 Transformation

The parenthesized predicate (p) is *true* if and only if p is *true*.

$$(p) \implies p$$

It is semantically equivalent to p.

C.6 Expression

C.6.1 Introduction

An `Expression` denotes a value in terms of the names with which values are associated by a model. An `Expression` can be any of schema universal quantification, schema existential quantification, schema unique existential quantification, function construction, definite description, substitution expression, schema equivalence, schema implication, schema disjunction, schema conjunction, schema negation, conditional, schema composition, schema piping, schema hiding, schema projection, schema precondition, Cartesian product, powerset, function and generic operator application, application, schema decoration, schema renaming, binding selection, tuple selection, binding construction, reference, generic instantiation, number literal, set extension, set comprehension, characteristic set comprehension, schema construction, binding extension, tuple extension, characteristic definite description, or parenthesized expression.

C.6.2 Schema universal quantification

C.6.2.1 Syntax

```
Expression      = ∀ , SchemaText , • , Expression
                | ...
                ;
```

C.6.2.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{⦂} \tau_1 \qquad \Sigma \oplus \beta_1 \vdash^{\varepsilon} e_2 \mathbin{⦂} \tau_2}{\Sigma \vdash^{\varepsilon} \forall (e_1 \mathbin{⦂} \tau_1) \bullet (e_2 \mathbin{⦂} \tau_2) \mathbin{⦂} \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[dom\ \beta_1 \lhd \beta_2] \end{array} \right)$$

In a schema universal quantification expression $\forall\ e_1 \bullet e_2$, expression e_1 shall be a schema, and expression e_2, in an environment overridden by the signature of schema e_1, shall also be a schema, and the signatures of these two schemas shall be compatible. The type of the whole expression is that of a schema whose signature is computed by subtracting from the signature of e_2 those pairs whose names are in the signature of e_1.

C.6.2.3 Semantics

The value of the schema universal quantification expression $\forall\ e_1 \bullet e_2$ is the set of bindings of schema e_2 restricted to exclude names that are in the signature of e_1, for all bindings of the schema e_1.

$$[\![\ \forall\ e_1 \bullet e_2 \mathbin{⦂} \mathbb{P}\,\tau\ ]\!]^{\varepsilon} \quad = \quad \lambda\ M : Model \bullet \{t_2 : [\![\ \tau\ ]\!]^{\tau} M \mid \forall\ t_1 : [\![\ e_1\ ]\!]^{\varepsilon} M \bullet t_1 \cup t_2 \in [\![\ e_2\ ]\!]^{\varepsilon}\ (M \oplus t_1) \bullet t_2\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the bindings (sets of pairs) in the semantic values of the carrier set of the type of the entire schema universal quantification expression in M, for which the union of the bindings (sets of pairs) in e_1 and in the whole expression is in the set that is the semantic value of e_2 in the model M overridden with the binding in e_1.

C.6.3 Schema existential quantification

C.6.3.1 Syntax

```
Expression      = ...
                | ∃ , SchemaText , • , Expression
                | ...
                ;
```

C.6.3.2 Transformation

The value of the schema existential quantification expression $\exists\ t \bullet e$ is the set of bindings of schema e restricted to exclude names that are in the signature of t, for at least one binding of the schema t.

$$\exists\ t \bullet e \quad \Longrightarrow \quad \neg\ \forall\ t \bullet \neg\ e$$

It is semantically equivalent to the result of applying de Morgan's law.

C.6.4 Schema unique existential quantification

C.6.4.1 Syntax

```
Expression      = ...
                | ∃1 , SchemaText , • , Expression
                | ...
                ;
```

C.6.4.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ ⦂\ \tau_1 \qquad \Sigma \oplus \beta_1 \vdash^{\varepsilon} e_2 \ ⦂\ \tau_2}{\Sigma \vdash^{\varepsilon} \exists_1\ (e_1 \ ⦂\ \tau_1) \bullet (e_2 \ ⦂\ \tau_2) \ ⦂\ \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[dom\ \beta_1 \lhd \beta_2] \end{array} \right)$$

In a schema unique existential quantification expression $\exists_1\ e_1 \bullet e_2$, expression e_1 shall be a schema, and expression e_2, in an environment overridden by the signature of schema e_1, shall also be a schema, and the signatures of these two schemas shall be compatible. The type of the whole expression is that of a schema whose signature is computed by subtracting from the signature of e_2 those pairs whose names are in the signature of e_1.

C.6.4.3 Semantics

The value of the schema unique existential quantification expression $\exists_1\ e_1 \bullet e_2$ is the set of bindings of schema e_2 restricted to exclude names that are in the signature of e_1, for at least one binding of the schema e_1.

$$\exists_1\ e_1 \bullet e_2 \quad \Longrightarrow \quad \neg\ (\forall\ e_1 \bullet \neg\ (e_2 \wedge (\forall\ [e_1 \mid e_2]^{\bowtie} \bullet \theta\ e_1 = \theta\ e_1\ ^{\bowtie})))$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$\exists_1\ e_1 \bullet e_2 \quad \Longrightarrow \quad \exists\ e_1 \bullet e_2 \wedge (\forall\ [e_1 \mid e_2]^{\bowtie} \bullet \theta\ e_1 = \theta\ e_1\ ^{\bowtie})$$

It is semantically equivalent to a schema existential quantification expression, analogous to the unique existential quantification predicate transformation.

C.6.5 Function construction

C.6.5.1 Syntax

```
Expression      = ...
                | λ , SchemaText , • , Expression
                | ...
                ;
```

C.6.5.2 Transformation

The value of the function construction expression $\lambda\ t \bullet e$ is the function associating values of the characteristic tuple of t with corresponding values of e.

$$\lambda\ t \bullet e \quad \Longrightarrow \quad \{t \bullet (chartuple\ t, e)\}$$

It is semantically equivalent to the set of pairs representation of that function.

C.6.6 Definite description

C.6.6.1 Syntax

```
Expression      = ...
                | μ , SchemaText , • , Expression
                | ...
                ;
```

C.6.6.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ ⨟\ \tau_1 \qquad \Sigma \oplus \beta \vdash^{\varepsilon} e_2 \ ⨟\ \tau_2}{\Sigma \vdash^{\varepsilon} \mu\,(e_1 \ ⨟\ \tau_1) \bullet (e_2 \ ⨟\ \tau_2) \ ⨟\ \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_3 = \tau_2 \end{array} \right)$$

In a definite description expression $\mu\ e_1 \bullet e_2$, expression e_1 shall be a schema. The type of the whole expression is the type of expression e_2, as determined in an environment overridden by the signature of schema e_1.

C.6.6.3 Semantics

The value of the definite description expression $\mu\ e_1 \bullet e_2$ is the sole value of e_2 that arises whichever binding is chosen from the set that is the value of schema e_1.

$$\begin{array}{l} \{M : Model;\ t_1 : \mathbb{W} \\ \quad \mid t_1 \in [\![\ e_1\ ]\!]^{\varepsilon} M \\ \quad \wedge\ (\forall\ t_3 : [\![\ e_1\ ]\!]^{\varepsilon} M \bullet [\![\ e_2\ ]\!]^{\varepsilon}\ (M \oplus t_3) = [\![\ e_2\ ]\!]^{\varepsilon}\ (M \oplus t_1)) \\ \bullet\ M \mapsto [\![\ e_2\ ]\!]^{\varepsilon}\ (M \oplus t_1)\} \end{array} \quad \subseteq \quad [\![\ \mu\ e_1 \bullet e_2\ ]\!]^{\varepsilon}$$

In terms of the semantic universe, its semantic value, given a model M in which the value of e_2 in that model overridden by a binding of the schema e_1 is the same regardless of which binding is chosen, is that value of e_2. In other models, it has a semantic value, but this loose definition of the semantics does not say what it is.

C.6.7 Substitution expression

C.6.7.1 Syntax

```
Expression      = ...
                | let , DeclName , == , Expression ,
                      { ;-tok , DeclName , == , Expression } ,
                          • , Expression
                | ...
                ;
```

C.6.7.2 Transformation

The value of the substitution expression let $i_1 == e_1$; ...; $i_n == e_n \bullet e$ is the value of e when all of its references to the names have been substituted by the values of the corresponding expressions.

$$\text{let } i_1 == e_1;\ ...;\ i_n == e_n \bullet e \quad \Longrightarrow \quad \mu\ i_1 == e_1;\ ...;\ i_n == i_n \bullet e$$

It is semantically equivalent to the similar definite description expression.

C.6.8 Schema equivalence

C.6.8.1 Syntax

```
Expression      = ...
                | Expression , ⇔ , Expression
                | ...
                ;
```

C.6.8.2 Transformation

The value of the schema equivalence expression $e_1 \Leftrightarrow e_2$ is that schema whose signature is the union of those of schemas e_1 and e_2, and whose bindings are those whose relevant restrictions are either both or neither in e_1 and e_2.

$$e_1 \Leftrightarrow e_2 \quad \Longrightarrow \quad (e_1 \Rightarrow e_2) \wedge (e_2 \Rightarrow e_1)$$

It is semantically equivalent to the schema conjunction of the schema implication $e_1 \Rightarrow e_2$ with the schema implication $e_2 \Rightarrow e_1$.

C.6.9 Schema implication

C.6.9.1 Syntax

```
Expression      = ...
                | Expression , ⇒ , Expression
                | ...
                ;
```

C.6.9.2 Transformation

The value of the schema implication expression $e_1 \Rightarrow e_2$ is that schema whose signature is the union of those of schemas e_1 and e_2, and whose bindings are those whose restriction to the signature of e_2 is in the value of e_2 if its restriction to the signature of e_1 is in the value of e_1.

$$e_1 \Rightarrow e_2 \quad \Longrightarrow \quad \neg\, e_1 \vee e_2$$

It is semantically equivalent to the schema disjunction of the schema negation $\neg\, e_1$ with e_2.

C.6.10 Schema disjunction

C.6.10.1 Syntax

```
Expression      = ...
                | Expression , ∨ , Expression
                | ...
                ;
```

C.6.10.2 Transformation

The value of the schema disjunction expression $e_1 \vee e_2$ is that schema whose signature is the union of those of schemas e_1 and e_2, and whose bindings are those whose restriction to the signature of e_1 is in the value of e_1 or its restriction to the signature of e_2 is in the value of e_2.

$$e_1 \vee e_2 \quad \Longrightarrow \quad \neg\,(\neg\, e_1 \wedge \neg\, e_2)$$

It is semantically equivalent to the schema negation of the schema conjunction of the schema negations of e_1 and e_2.

C.6.11 Schema conjunction

C.6.11.1 Syntax

```
Expression      = ...
                | Expression , ∧ , Expression
                | ...
                ;
```

C.6.11.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \,:\!\!\circ\, \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \,:\!\!\circ\, \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \,:\!\!\circ\, \tau_1) \wedge (e_2 \,:\!\!\circ\, \tau_2) \,:\!\!\circ\, \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_1 \approx \beta_2 \\ \tau_3 = \mathbb{P}[\beta_1 \cup \beta_2] \end{array} \right)$$

In a schema conjunction expression $e_1 \wedge e_2$, expressions e_1 and e_2 shall be schemas, and their signatures shall be compatible. The type of the whole expression is that of the schema whose signature is the union of those of expressions e_1 and e_2.

C.6.11.3 Semantics

The value of the schema conjunction expression $e_1 \wedge e_2$ is the schema resulting from merging the signatures of schemas e_1 and e_2 and conjoining their constraints.

$$[\![\, e_1 \wedge e_2 \,:\!\!\circ\, \mathbb{P}\,\tau \,]\!]^{\mathcal{E}} \;=\; \lambda\, M : \mathit{Model} \bullet \{ t : [\![\, \tau \,]\!]^{\mathcal{T}} M;\; t_1 : [\![\, e_1 \,]\!]^{\mathcal{E}} M;\; t_2 : [\![\, e_2 \,]\!]^{\mathcal{E}} M \mid t_1 \cup t_2 = t \bullet t \}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the unions of the bindings (sets of pairs) in the semantic values of e_1 and e_2 in M.

C.6.12 Schema negation

C.6.12.1 Syntax

```
Expression      = ...
                | ¬ , Expression
                | ...
                ;
```

C.6.12.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \ : \ \tau_1}{\Sigma \vdash^{\mathcal{E}} \neg\,(e \ : \ \tau_1) \ : \ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = \tau_1 \end{array} \right)$$

In a schema negation expression $\neg\ e$, expression e shall be a schema. The type of the whole expression is the same as the type of expression e.

C.6.12.3 Semantics

The value of the schema negation expression $\neg\ e$ is that set of bindings that are of the same type as those in schema e but that are not in schema e.

$$[\![\ \neg\ e \ : \ \mathbb{P}\,\tau\ ]\!]^{\mathcal{E}} \ = \ \lambda\ M : \mathit{Model} \bullet \{t : [\![\ \tau\ ]\!]^{\mathcal{T}} M \mid \neg\ t \in [\![\ e\ ]\!]^{\mathcal{E}} M \bullet t\}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the bindings (sets of pairs) that are members of the semantic value of the carrier set of schema e in M such that those bindings are not members of the semantic value of schema e in M.

C.6.13 Conditional

C.6.13.1 Syntax

```
Expression      = ...
                | if , Predicate , then , Expression , else , Expression
                | ...
                ;
```

C.6.13.2 Transformation

The value of the conditional expression if p then e_1 else e_2 is the value of e_1 if p is *true*, and is the value of e_2 if p is *false*.

$$\text{if } p \text{ then } e_1 \text{ else } e_2 \quad \Longrightarrow \quad \mu\ i : \{e_1, e_2\} \mid p \wedge i = e_1 \vee \neg\ p \wedge i = e_2 \bullet i$$

It is semantically equivalent to the definite description expression whose value is either that of e_1 or that of e_2 such that if p is *true* then it is e_1 or if p is *false* then it is e_2.

C.6.14 Schema composition

C.6.14.1 Syntax

```
Expression      = ...
                | Expression , ⨾ , Expression
                | ...
                ;
```


C.6.14.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \ : \ \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \ : \ \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \ : \ \tau_1) \fatsemi (e_2 \ : \ \tau_2) \ : \ \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ match = \{i : dom\ \beta_2 \mid i\ decor\ ' \in dom\ \beta_1 \bullet i\} \\ \beta_3 = \{i : match \bullet i\ decor\ '\} \lhd \beta_1 \\ \beta_4 = match \lhd \beta_2 \\ \beta_3 \approx \beta_4 \\ \{i : match \bullet i \mapsto \beta_1(i\ decor\ ')\} \approx \{i : match \bullet i \mapsto \beta_2\ i\} \\ \tau_3 = \mathbb{P}[\beta_3 \cup \beta_4] \end{array} \right)$$

In a schema composition expression $e_1 \fatsemi e_2$, expressions e_1 and e_2 shall be schemas. Let *match* be the set of names in schema e_2 for which there are matching primed names in schema e_1. Let β_3 be the signature formed from the components of e_1 excluding the matched primed components. Let β_4 be the signature formed from the components of e_2 excluding the matched unprimed components. Signatures β_3 and β_4 shall be compatible. The types of the excluded matched pairs of components shall be the same. The type of the whole expression is that of a schema whose signature is the union of β_3 and β_4.

C.6.14.3 Semantics

The value of the schema composition expression $e_1 \fatsemi e_2$ is that schema representing the operation of doing the operations represented by schemas e_1 and e_2 in sequence.

$$\begin{array}{l} (e_1 \ : \ \mathbb{P}[\sigma_1]) \fatsemi (e_2 \ : \ \mathbb{P}[\sigma_2]) \ : \ \mathbb{P}[\sigma] \\ \Longrightarrow \\ \neg\ (\forall\ e^{\bowtie} \bullet \neg\ (\neg\ (\forall\ e_3 \bullet \neg\ [e_1;\ e^{\bowtie} \mid \theta\ e_3 = \theta\ e^{\bowtie}]) \\ \qquad \land \neg\ (\forall\ e_4 \bullet \neg\ [e_2;\ e^{\bowtie} \mid \theta\ e_4 = \theta\ e^{\bowtie}]))) \\ where\ e_3 == carrier\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i\ decor\ ' \mapsto \tau \in \sigma_1 \bullet i\ decor\ ' \mapsto \tau\}] \\ \quad and\ e_4 == carrier\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i \mapsto \tau \in \sigma_2 \bullet i \mapsto \tau\}] \\ \quad and\ e^{\bowtie} == (e_4)^{\bowtie} \end{array}$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$\begin{array}{l} (e_1 \ : \ \mathbb{P}[\sigma_1]) \fatsemi (e_2 \ : \ \mathbb{P}[\sigma_2]) \ : \ \mathbb{P}[\sigma] \\ \Longrightarrow \\ \exists\ e^{\bowtie} \bullet (\exists\ e_3 \bullet [e_1;\ e^{\bowtie} \mid \theta\ e_3 = \theta\ e^{\bowtie}]) \\ \qquad \land (\exists\ e_4 \bullet [e_2;\ e^{\bowtie} \mid \theta\ e_4 = \theta\ e^{\bowtie}]) \\ where\ e_3 == carrier\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i\ decor\ ' \mapsto \tau \in \sigma_1 \bullet i\ decor\ ' \mapsto \tau\}] \\ \quad and\ e_4 == carrier\ \ [\{i : \mathtt{NAME};\ \tau : \mathtt{Type} \mid i \mapsto \tau \in \sigma_2 \bullet i \mapsto \tau\}] \\ \quad and\ e^{\bowtie} == (e_4)^{\bowtie} \end{array}$$

It is semantically equivalent to the existential quantification of the matched pairs of primed components of e_1 and unprimed components of e_2, with those matched pairs being equated.

C.6.15 Schema piping

C.6.15.1 Syntax

```
Expression      = ...
                | Expression , ≫ , Expression
                | ...
                ;
```


C.6.15.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \ : \ \tau_1 \qquad \Sigma \vdash^{\mathcal{E}} e_2 \ : \ \tau_2}{\Sigma \vdash^{\mathcal{E}} (e_1 \ : \ \tau_1) \gg (e_2 \ : \ \tau_2) \ : \ \tau_3} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta_1] \\ \tau_2 = \mathbb{P}[\beta_2] \\ match = \{i : \mathtt{NAME} \mid i \ decor \ ! \in dom \ \beta_1 \wedge i \ decor \ ? \in dom \ \beta_2 \bullet i\} \\ \beta_3 = \{i : match \bullet i \ decor \ !\} \lhd \beta_1 \\ \beta_4 = \{i : match \bullet i \ decor \ ?\} \lhd \beta_2 \\ \beta_3 \approx \beta_4 \\ \{i : match \bullet i \mapsto \beta_1(i \ decor \ !)\} \approx \{i : match \bullet i \mapsto \beta_2 \ (i \ decor \ ?)\} \\ \tau_3 = \mathbb{P}[\beta_3 \cup \beta_4] \end{array} \right)$$

In a schema piping expression $e_1 \gg e_2$, expressions e_1 and e_2 shall be schemas. Let *match* be the set of names for which there are matching shrieked names in schema e_1 and queried names in schema e_2. Let β_3 be the signature formed from the components of e_1 excluding the matched shrieked components. Let β_4 be the signature formed from the components of e_2 excluding the matched queried components. Signatures β_3 and β_4 shall be compatible. The types of the excluded matched pairs of components shall be the same. The type of the whole expression is that of a schema whose signature is the union of β_3 and β_4.

C.6.15.3 Semantics

The value of the schema piping expression $e_1 \gg e_2$ is that schema representing the operation formed from the two operations represented by schemas e_1 and e_2 with the outputs of e_1 identified with the inputs of e_2.

$$\begin{array}{l} (e_1 \ : \ \mathbb{P}[\sigma_1]) \gg (e_2 \ : \ \mathbb{P}[\sigma_2]) \ : \ \mathbb{P}[\sigma] \\ \qquad\qquad \Longrightarrow \\ \neg \ (\forall \ e^{\bowtie} \bullet \neg \ (\neg \ (\forall \ e_3 \bullet \neg \ [e_1; \ e^{\bowtie} \mid \theta \ e_3 = \theta \ e^{\bowtie}]) \\ \qquad\qquad \wedge \neg \ (\forall \ e_4 \bullet \neg \ [e_2; \ e^{\bowtie} \mid \theta \ e_4 = \theta \ e^{\bowtie}]))) \\ where \ e_3 == carrier \ \ [\{i : \mathtt{NAME}; \ \tau : \mathtt{Type} \mid i \ decor \ ! \mapsto \tau \in \sigma_1 \bullet i \ decor \ ! \mapsto \tau\}] \\ \quad and \ e_4 == carrier \ \ [\{i : \mathtt{NAME}; \ \tau : \mathtt{Type} \mid i \ decor \ ? \mapsto \tau \in \sigma_2 \bullet i \ decor \ ? \mapsto \tau\}] \\ \quad and \ e^{\bowtie} == (e_4)^{\bowtie} \end{array}$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$\begin{array}{l} (e_1 \ : \ \mathbb{P}[\sigma_1]) \gg (e_2 \ : \ \mathbb{P}[\sigma_2]) \ : \ \mathbb{P}[\sigma] \\ \qquad\qquad \Longrightarrow \\ \exists \ e^{\bowtie} \bullet (\exists \ e_3 \bullet [e_1; \ e^{\bowtie} \mid \theta \ e_3 = \theta \ e^{\bowtie}]) \\ \qquad \wedge (\exists \ e_4 \bullet [e_2; \ e^{\bowtie} \mid \theta \ e_4 = \theta \ e^{\bowtie}]) \\ where \ e_3 == carrier \ \ [\{i : \mathtt{NAME}; \ \tau : \mathtt{Type} \mid i \ decor \ ! \mapsto \tau \in \sigma_1 \bullet i \ decor \ ! \mapsto \tau\}] \\ \quad and \ e_4 == carrier \ \ [\{i : \mathtt{NAME}; \ \tau : \mathtt{Type} \mid i \ decor \ ? \mapsto \tau \in \sigma_2 \bullet i \ decor \ ? \mapsto \tau\}] \\ \quad and \ e^{\bowtie} == (e_4)^{\bowtie} \end{array}$$

It is semantically equivalent to the existential quantification of the matched pairs of shrieked components of e_1 and queried components of e_2, with those matched pairs being equated.

C.6.16 Schema hiding

C.6.16.1 Syntax

```
Expression      = ...
                | Expression , \ , (-tok , DeclName , { ,-tok , DeclName } , )-tok
                | ...
                ;
```


C.6.16.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{:} \tau_1}{\Sigma \vdash^{\varepsilon} (e \mathbin{:} \tau_1) \setminus (i_1, ..., i_n) \mathbin{:} \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \{i_1, \ ..., \ i_n\} \subseteq dom \ \beta \\ \tau_2 = \mathbb{P}[\{i_1, \ ..., \ i_n\} \lhd \beta] \end{array} \right)$$

In a schema hiding expression $e \setminus (i_1, ..., i_n)$, expression e shall be a schema, and the names to be hidden shall all be in the signature of that schema. The type of the whole expression is that of a schema whose signature is computed by subtracting from the signature of expression e those pairs whose names are hidden.

C.6.16.3 Semantics

The value of the schema hiding expression $e \setminus (i_1, ..., i_n)$ is that schema whose signature is that of schema e minus the hidden names, and whose bindings have the same values as those in schema e.

$$(e \mathbin{:} \mathbb{P}[\sigma]) \setminus (i_1, ..., i_n) \implies \neg \ (\forall \ i_1 : carrier \ (\sigma \ i_1); \ ...; \ i_n : carrier \ (\sigma \ i_n) \bullet \neg \ e)$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$(e \mathbin{:} \mathbb{P}[\sigma]) \setminus (i_1, ..., i_n) \implies \exists \ i_1 : carrier \ (\sigma \ i_1); \ ...; \ i_n : carrier \ (\sigma \ i_n) \bullet e$$

It is semantically equivalent to the schema existential quantification of the hidden names $i_1, ..., i_n$ from the schema e.

C.6.17 Schema projection

C.6.17.1 Syntax

```
Expression      = ...
                | Expression , ⌈ , Expression
                | ...
                ;
```

C.6.17.2 Transformation

The value of the schema projection expression $e_1 \upharpoonright e_2$ is the schema that is like the conjunction $e_1 \wedge e_2$ but whose signature is restricted to just that of schema e_2.

$$e_1 \upharpoonright e_2 \implies \{e_1; \ e_2 \bullet \theta \ e_2\}$$

It is semantically equivalent to that set of bindings of names in the signature of e_2 to values that satisfy the constraints of both e_1 and e_2.

C.6.18 Schema precondition

C.6.18.1 Syntax

```
Expression      = ...
                | pre , Expression
                | ...
                ;
```


C.6.18.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \ ⁝ \ \tau_1}{\Sigma \vdash^{\mathcal{E}} \text{pre}\ (e \ ⁝ \ \tau_1) \ ⁝ \ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = \mathbb{P}[\{i, j : \texttt{NAME} \mid j \in dom\ \beta \wedge (j = i\ decor\ ' \vee j = i\ decor\ !) \bullet j\} \lhd \beta] \end{array} \right)$$

In a schema precondition expression pre e, expression e shall be a schema. The type of the whole expression is that of a schema whose signature is computed by subtracting from the signature of e those pairs whose names have primed or shrieked decorations.

C.6.18.3 Semantics

The value of the schema precondition expression pre e is that schema which is like schema e but without its primed and shrieked components.

$$\text{pre}(e \ ⁝ \ \mathbb{P}[\sigma_1]) \ ⁝ \ \mathbb{P}[\sigma_2] \quad \Longrightarrow \quad \neg\ (\forall\ carrier\ [\sigma_1 \setminus \sigma_2] \bullet \neg\ e)$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$\text{pre}(e \ ⁝ \ \mathbb{P}[\sigma_1]) \ ⁝ \ \mathbb{P}[\sigma_2] \quad \Longrightarrow \quad \exists\ carrier\ [\sigma_1 \setminus \sigma_2] \bullet e$$

It is semantically equivalent to the existential quantification of the primed and shrieked components from the schema e.

C.6.19 Cartesian product

C.6.19.1 Syntax

```
Expression      = ...
                | Expression , × , Expression , { × , Expression }
                | ...
                ;
```

C.6.19.2 Transformation

The value of the Cartesian product expression $e_1 \times ... \times e_n$ is the set of all tuples whose components are members of the corresponding sets that are the values of its expressions.

$$e_1 \times ... \times e_n \quad \Longrightarrow \quad \{i_1 : e_1;\ ...;\ i_n : e_n \bullet (i_1, ..., i_n)\}$$

It is semantically equivalent to the set comprehension expression that declares members of the sets and assembles those members into tuples.

C.6.20 Powerset

C.6.20.1 Syntax

```
Expression      = ...
                | ℙ , Expression
                | ...
                ;
```


C.6.20.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \ : \ \tau_1}{\Sigma \vdash^{\mathcal{E}} \mathbb{P}(e \ : \ \tau_1) \ : \ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}\,\alpha \\ \tau_2 = \mathbb{P}\,\tau_1 \end{array} \right)$$

In a powerset expression $\mathbb{P}\,e$, expression e shall be a set. The type of the whole expression is then a powerset of the type of expression e.

C.6.20.3 Semantics

The value of the powerset expression $\mathbb{P}\,e$ is the set of all subsets of the set that is the value of e.

$$[\![\ \mathbb{P}\,e\ ]\!]^{\mathcal{E}} \ = \ \lambda\ M : \mathit{Model} \bullet \mathbb{P}\,([\![\ e\ ]\!]^{\mathcal{E}} M)$$

In terms of the semantic universe, its semantic value, given a model M, is the powerset of values of e in M.

C.6.21 Function and generic operator application

C.6.21.1 Syntax

```
Expression      = ...
                | Application
                | ...
                ;

Application     = PrefixApp
                | PostfixApp
                | InfixApp
                | NofixApp
                ;

PrefixApp       = PRE , Expression
                | L , ExpSep , ( Expression , ERE | ExpressionList , SRE ) , Expression
                ;

PostfixApp      = Expression , POST
                | Expression , EL , ExpSep , ( Expression , ER | ExpressionList , SR )
                ;

InfixApp        = Expression , I , Expression
                | Expression , EL , ExpSep ,
                    ( Expression , ERE | ExpressionList , SRE ) , Expression
                ;

NofixApp        = L , ExpSep , ( Expression , ER | ExpressionList , SR ) ;
```

C.6.21.2 Transformation

All function operator applications are transformed to annotated application expressions.

All generic operator applications are transformed to annotated generic instantiation expressions.

For any particular function or generic operator application, two potential transformations are specified below, both of which result in the same `NAME`. That `NAME` shall be one for which there is an operator template paragraph in scope (see 12.2.8). Which of the two transformations is appropriate is determined by that operator template's category: function or generic respectively.

The left-hand sides of many of these transformation rules involve `ExpSep` phrases: they use *es* metavariables. None of them use *ss* metavariables: in other words, only the `Expression ES` case of `ExpSep` is specified, not the `ExpressionList SS` case. Where the latter case occurs in a specification, the `ExpressionList` shall be transformed by the rule in 12.2.12 to an expression, and thence a transformation analogous to that specified for the former case can be performed, differing only in that a *ss* appears in the function or generic name in place of an *es*.

C.6.21.2.1 `PrefixApp`

Function cases

$$\begin{aligned} pre\ e &\Longrightarrow pre\bowtie\ e \\ ln\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ ere\ e_n &\Longrightarrow ln\bowtie es_1 \ldots \bowtie es_{n-2}\bowtie ere\bowtie\ (e_1, \ldots, e_{n-2}, e_{n-1}, e_n) \\ ln\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ sre\ e_n &\Longrightarrow ln\bowtie es_1 \ldots \bowtie es_{n-2}\bowtie sre\bowtie\ (e_1, \ldots, e_{n-2}, a_{n-1}, e_n) \end{aligned}$$

Generic cases

$$\begin{aligned} pre\ e &\Longrightarrow pre\bowtie\ [e] \\ ln\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ ere\ e_n &\Longrightarrow ln\bowtie es_1 \ldots \bowtie es_{n-2}\bowtie ere\bowtie\ [e_1, \ldots, e_{n-2}, e_{n-1}, e_n] \\ ln\ e_1\ es_1\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ sre\ e_n &\Longrightarrow ln\bowtie es_1 \ldots \bowtie es_{n-2}\bowtie sre\bowtie\ [e_1, \ldots, e_{n-2}, a_{n-1}, e_n] \end{aligned}$$

C.6.21.2.2 `PostfixApp`

Function cases

$$\begin{aligned} e\ post &\Longrightarrow \bowtie post\ e \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ e_n\ er &\Longrightarrow \bowtie el\bowtie es_2 \ldots \bowtie es_{n-1}\bowtie er\ (e_1, e_2, \ldots, e_{n-1}, e_n) \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ a_n\ sr &\Longrightarrow \bowtie el\bowtie es_2 \ldots \bowtie es_{n-1}\bowtie sr\ (e_1, e_2, \ldots, e_{n-1}, a_n) \end{aligned}$$

Generic cases

$$\begin{aligned} e\ post &\Longrightarrow \bowtie post\ [e] \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ e_n\ er &\Longrightarrow \bowtie el\bowtie es_2 \ldots \bowtie es_{n-1}\bowtie er\ [e_1, e_2, \ldots, e_{n-1}, e_n] \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-1}\ es_{n-1}\ a_n\ sr &\Longrightarrow \bowtie el\bowtie es_2 \ldots \bowtie es_{n-1}\bowtie sr\ [e_1, e_2, \ldots, e_{n-1}, a_n] \end{aligned}$$

C.6.21.2.3 `InfixApp`

Function cases

$$\begin{aligned} e_1\ in\ e_2 &\Longrightarrow \bowtie in\bowtie\ (e_1, e_2) \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ ere\ e_n &\Longrightarrow \bowtie el\bowtie es_2 \ldots \bowtie es_{n-2}\bowtie ere\bowtie\ (e_1, e_2, \ldots, e_{n-2}, e_{n-1}, e_n) \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ sre\ e_n &\Longrightarrow \bowtie el\bowtie es_2 \ldots \bowtie es_{n-2}\bowtie sre\bowtie\ (e_1, e_2, \ldots, e_{n-2}, a_{n-1}, e_n) \end{aligned}$$

Generic cases

$$\begin{aligned} e_1\ in\ e_2 &\Longrightarrow \bowtie in\bowtie\ [e_1, e_2] \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-2}\ es_{n-2}\ e_{n-1}\ ere\ e_n &\Longrightarrow \bowtie el\bowtie es_2 \ldots \bowtie es_{n-2}\bowtie ere\bowtie\ [e_1, e_2, \ldots, e_{n-2}, e_{n-1}, e_n] \\ e_1\ el\ e_2\ es_2\ \ldots\ e_{n-2}\ es_{n-2}\ a_{n-1}\ sre\ e_n &\Longrightarrow \bowtie el\bowtie es_2 \ldots \bowtie es_{n-2}\bowtie sre\bowtie\ [e_1, e_2, \ldots, e_{n-2}, a_{n-1}, e_n] \end{aligned}$$

C.6.21.2.4 NofixApp

Function cases

$$ln\ e_1\ es_1\ \ldots\ e_{n-1}\ es_{n-1}\ e_n\ er \implies ln{\bowtie}es_1{\ldots}{\bowtie}es_{n-1}{\bowtie}er\ (e_1, \ldots, e_{n-1}, e_n)$$
$$ln\ e_1\ es_1\ \ldots\ e_{n-1}\ es_{n-1}\ a_n\ sr \implies ln{\bowtie}es_1{\ldots}{\bowtie}es_{n-1}{\bowtie}sr\ (e_1, \ldots, e_{n-1}, a_n)$$

Generic cases

$$ln\ e_1\ es_1\ \ldots\ e_{n-1}\ es_{n-1}\ e_n\ er \implies ln{\bowtie}es_1{\ldots}{\bowtie}es_{n-1}{\bowtie}er\ [e_1, \ldots, e_{n-1}, e_n]$$
$$ln\ e_1\ es_1\ \ldots\ e_{n-1}\ es_{n-1}\ a_n\ sr \implies ln{\bowtie}es_1{\ldots}{\bowtie}es_{n-1}{\bowtie}sr\ [e_1, \ldots, e_{n-1}, a_n]$$

C.6.21.3 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \,:\, \tau_1}{\Sigma \vdash^{\varepsilon} \mathbb{P}(e \,:\, \tau_1) \,:\, \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}\,\alpha \\ \tau_2 = \mathbb{P}\,\tau_1 \end{array} \right)$$

In a powerset expression $\mathbb{P}\,e$, expression e shall be a set. The type of the whole expression is then a powerset of the type of expression e.

C.6.21.4 Semantics

The value of the powerset expression $\mathbb{P}\,e$ is the set of all subsets of the set that is the value of e.

$$[\![\ \mathbb{P}\,e\ ]\!]^{\varepsilon} \;=\; \lambda\ M : Model \bullet \mathbb{P}\,([\![\ e\ ]\!]^{\varepsilon} M)$$

In terms of the semantic universe, its semantic value, given a model M, is the powerset of values of e in M.

C.6.22 Application

C.6.22.1 Syntax

```
Expression      = ...
                | Expression , Expression
                | ...
                ;
```

C.6.22.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \,:\, \tau_1 \qquad \Sigma \vdash^{\varepsilon} e_2 \,:\, \tau_2}{\Sigma \vdash^{\varepsilon} (e_1 \,:\, \tau_1)\ (e_2 \,:\, \tau_2) \,:\, \tau_3} \left(\tau_1 = \mathbb{P}(\tau_2 \times \tau_3) \right)$$

In an application expression $e_1\ e_2$, the expression e_1 shall be a set of pairs, and expression e_2 shall be of the same type as the first components of those pairs. The type of the whole expression is the type of the second components of those pairs.

C.6.22.3 Semantics

The value of the application expression $e_1\ e_2$ is the sole value associated with e_2 in the relation e_1.

$$e_1\ e_2 \,:\, \tau \implies (\mu\ i : carrier\ \tau \mid (e_2, i) \in e_1 \bullet i)$$

It is semantically equivalent to that sole range value i such that the pair (e_2, i) is in the set of pairs that is the value of e_1. If there is no value or more than one value associated with e_2, then the application expression has a value but what it is is not specified.

C.6.23 Schema decoration

C.6.23.1 Syntax

```
Expression      = ...
                | Expression , STROKE
                | ...
                ;
```

C.6.23.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \ ⦂ \ \tau_1}{\Sigma \vdash^{\varepsilon} (e \ ⦂ \ \tau_1)^{+} \ ⦂ \ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \tau_2 = \mathbb{P}[\{i : dom \ \beta \bullet i \ decor^{+} \mapsto \beta \ i\}] \end{array} \right)$$

In a schema decoration expression e^{+}, expression e shall be a schema. The type of the whole expression is that of a schema whose signature is like that of e but with the stroke appended to each of its names.

C.6.23.3 Semantics

The value of the schema decoration expression e^{+} is that schema whose bindings are like those of the schema e except that their names have the addition stroke $^{+}$.

$$(e \ ⦂ \ \mathbb{P}[i_1 : \tau_1; \ ...; \ i_n : \tau_n])^{+} \quad \Longrightarrow \quad e \ [i_1 \ decor^{+} / i_1, ..., i_n \ decor^{+} / i_n]$$

It is semantically equivalent to the schema renaming where decorated names rename the original names.

C.6.24 Schema renaming

C.6.24.1 Syntax

```
Expression      = ...
                | Expression , [-tok , DeclName , / , DeclName ,
                      { ,-tok , DeclName , / , DeclName } , ]-tok
                | ...
                ;
```

C.6.24.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \ ⦂ \ \tau_1}{\Sigma \vdash^{\varepsilon} (e \ ⦂ \ \tau_1)[j_1 / i_1, ..., j_n / i_n] \ ⦂ \ \tau_2} \left(\begin{array}{l} \# \{i_1, \ ..., \ i_n\} = n \\ \tau_1 = \mathbb{P}[\beta_1] \\ \beta_2 = \{j_1 \mapsto i_1, \ ..., \ j_n \mapsto i_n\} \fatsemi \beta_1 \cup \{i_1, \ ..., \ i_n\} \lhd \beta_1 \\ \tau_2 = \mathbb{P}[\beta_2] \\ \beta_2 \in (\_ \pfun \_) \end{array} \right)$$

In a schema renaming expression $e \ [j_1 \ / \ i_1, ..., j_n \ / \ i_n]$, there shall be no duplicates amongst the old names $i_1, ..., i_n$. Expression e shall be a schema. The type of the whole expression is that of a schema whose signature is like that of expression e but with the new names in place of corresponding old names. Declarations that are merged by the renaming shall have the same type.

NOTE Old names need not be in the signature of the schema. This is so as to permit renaming to distribute over other notations such as disjunction.

C.6.24.3 Semantics

The value of the schema renaming expression $e\ [j_1\ /\ i_1, ..., j_n\ /\ i_n]$ is that schema whose bindings are like those of schema e except that some of its names have been replaced by new names, possibly merging components.

$$\begin{aligned}[\![\ e\ [j_1\ /\ i_1, ..., j_n\ /\ i_n]\ ]\!]^{\mathcal{E}} \quad = \quad & \lambda\ M : Model \bullet \\ & \{t_1 : [\![\ e\ ]\!]^{\mathcal{E}} M;\ t_2 : \mathbb{W} \mid \\ & \quad t_2 = \{j_1 \mapsto i_1, ..., j_n \mapsto i_n\} \fatsemi t_1 \cup \{i_1, ..., i_n\} \lhd t_1 \\ & \quad \land\ t_2 \in (\_ \pfun \_) \\ & \bullet\ t_2\}\end{aligned}$$

In terms of the semantic universe, its semantic value, given a model M, is the set of the bindings (sets of pairs) in the semantic value of e in M with the new names replacing corresponding old names. Where components are merged by the renaming, those components shall have the same value.

C.6.25 Binding selection

C.6.25.1 Syntax

```
Expression      = ...
                | Expression , . , RefName
                | ...
                ;
```

C.6.25.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \fatsemi \tau_1}{\Sigma \vdash^{\mathcal{E}} (e \fatsemi \tau_1)\ .\ i \fatsemi \tau_2} \left(\begin{array}{l} \tau_1 = [\beta] \\ (i,\ \tau_2) \in \beta \end{array} \right)$$

In a binding selection expression $e\ .\ i$, expression e shall be a binding, and name i shall select one of its components. The type of the whole expression is the type of the selected component.

C.6.25.3 Semantics

The value of the binding selection expression $e\ .\ i$ is that value associated with i in the binding that is the value of e.

$$(e \fatsemi [\sigma])\ .\ i \quad \Longrightarrow \quad \{carrier\ [\sigma] \bullet (chartuple\ (carrier\ \ [\sigma]), i)\}\ e$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$(e \fatsemi [\sigma])\ .\ i \quad \Longrightarrow \quad (\lambda\ carrier\ [\sigma] \bullet i)\ e$$

It is semantically equivalent to the function construction expression, from bindings of the schema type of e, to the value of the selected name i, applied to the particular binding e.

C.6.26 Tuple selection

C.6.26.1 Syntax

```
Expression      = ...
                | Expression , . , NUMERAL
                | ...
                ;
```

C.6.26.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \ : \ \tau_1}{\Sigma \vdash^{\varepsilon} (e \ : \ \tau_1) \, . \, b \ : \ \tau_2} \left(\begin{array}{l} \tau_1 = \alpha_1 \times ... \times \alpha_k \\ b \in 1 \, .. \, k \\ \tau_2 = \alpha_b \end{array} \right)$$

In a tuple selection expression $e \, . \, b$, the type of expression e shall be a Cartesian product, and the numeric value of NUMERAL b shall select one of its components. The type of the whole expression is the type of the selected component.

C.6.26.3 Semantics

The value of the tuple selection expression $e \, . \, b$ is the b'th component of the tuple that is the value of e.

$$(e \ : \ \tau_1 \times ... \times \tau_n) \, . \, b \implies \{i : carrier \, (\tau_1 \times ... \times \tau_n) \bullet (i, \mu \ i_1 : carrier \ \tau_1; \ ...; \ i_n : carrier \ \tau_n \mid i = (i_1, ..., i_n) \bullet i_b)\} \ e$$

NOTE Exploiting notation that is not present in the annotated syntax, this abbreviates to the following.

$$(e \ : \ \tau_1 \times ... \times \tau_n) \, . \, b \implies (\lambda \ i : carrier \, (\tau_1 \times ... \times \tau_n) \bullet \mu \ i_1 : carrier \ \tau_1; \ ...; \ i_n : carrier \ \tau_n \mid i = (i_1, ..., i_n) \bullet i_b) \ e$$

It is semantically equivalent to the function construction, from tuples of the Cartesian product type to the selected component of the tuple b, applied to the particular tuple e.

C.6.27 Binding construction

C.6.27.1 Syntax

```
Expression      = ...
                | θ , Expression , { STROKE }
                | ...
                ;
```

C.6.27.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \ : \ \tau_1}{\Sigma \vdash^{\varepsilon} \theta \, (e \ : \ \tau_1) \ ^* \ : \ \tau_2} \left(\begin{array}{l} \tau_1 = \mathbb{P}[\beta] \\ \forall \ i : dom \ \beta \bullet (i \ decor \ ^*, \beta \ i) \in \Sigma \wedge \neg \ generic\_type \ (\beta \ i) \\ \tau_2 = [\beta] \end{array} \right)$$

In a binding construction expression $\theta \, e \ ^*$, the expression e shall be a schema. Every name and type pair in its signature, with the optional decoration added, shall be present in the environment with a non-generic type. The type of the whole expression is that of a binding whose signature is that of the schema.

NOTE If the type in the environment were generic, the semantic transformation in 14.2.5.2 would produce a reference expression whose implicit instantiation is not determined by this International Standard.

C.6.27.3 Semantics

The value of the binding construction expression $\theta\, e\, ^*$ is the binding whose names are those in the signature of schema e and whose values are those of the same names with the optional decoration appended.

$$\theta\, e\, ^*\ :\ [i_1 : \tau_1;\ ...;\ i_n : \tau_n] \quad \Longrightarrow \quad \langle\!|\ i_1 == i_1\ decor\ ^*, ..., i_n == i_n\ decor\ ^*\ |\!\rangle$$

It is semantically equivalent to the binding extension expression whose value is that binding.

C.6.28 Reference

C.6.28.1 Syntax

```
Expression      = ...
                | RefName
                | ...
                ;
```

C.6.28.2 Type

In a reference expression, if the name is of the form Δi and no declaration of this name yet appears in the environment, then the following syntactic transformation is applied.

$$\Delta i \quad \overset{\Delta i \notin dom\ \Sigma}{\Longrightarrow} \quad [i;\ i\ ']$$

This syntactic transformation makes the otherwise undefined name be equivalent to the corresponding schema construction expression, which is then typechecked.

Similarly, if the name is of the form Ξi and no declaration of this name yet appears in the environment, then the following syntactic transformation is applied.

$$\Xi i \quad \overset{\Xi i \notin dom\ \Sigma}{\Longrightarrow} \quad [i;\ i\ '\mid \theta\, i = \theta\, i\ ']$$

NOTE 1 The Ξ notation is deliberately not defined in terms of the Δ notation.

NOTE 2 Type inference could be done without these syntactic transformations, but they are necessary steps in defining the formal semantics.

NOTE 3 Only occurrences of Δ and Ξ that are in such reference expressions are so transformed, not others such as those in the names of declarations.

$$\frac{\left(\begin{array}{l} i \in dom\ \Sigma \\ \tau = \text{if } generic\_type\ (\Sigma\ i) \text{ then } \Sigma\ i, (\Sigma\ i)\ [\alpha_1, ..., \alpha_n] \text{ else } \Sigma\ i \end{array}\right)}{\Sigma\ \vdash^{\varepsilon}\ i : \tau}$$

In any other reference expression i, the name i shall be associated with a type in the environment. If that type is generic, the annotation of the whole expression is a pair of both the uninstantiated type (for the Instantiation clause to determine that this is a reference to a generic definition) and the type instantiated with new distinct type variables (which the context shall constrain to non-generic types). Otherwise (if the type in the environment is non-generic), that is the type of the whole expression.

NOTE 4 If the type is generic, the reference expression will be transformed to a generic instantiation expression by the rule in 13.2.3.3. That shall be done only when the implicit instantiations have been determined via constraints on the new type variables $\alpha_1, ..., \alpha_n$.

C.6.28.3 Semantics

The value of a reference expression that refers to a generic definition is an inferred instantiation of that generic definition.

$$i \mathrel{⦂} [i_1, ..., i_n]\tau, \tau' \quad \tau' = ([i_1, ..., i_n]\tau) \Longrightarrow [\alpha_1, ..., \alpha_n] \quad i\ [carrier\ \alpha_1, ..., carrier\ \alpha_n] \mathrel{⦂} \tau'$$

It is semantically equivalent to the generic instantiation expression whose generic actuals are the carrier sets of the types inferred for the generic parameters. The type τ' is an instantiation of the generic type τ. The types inferred for the generic parameters are α_1, ..., α_n. They shall all be determinable by comparison of τ with τ' as suggested by the condition on the transformation. Cases where these types cannot be so determined, because the generic type is independent of some of the generic parameters, are not well-typed.

EXAMPLE 1 The paragraph

$a[X] == 1$

defines a with type $[X]\texttt{GIVEN}\ \mathbb{A}$. The paragraph

$b == a$

typechecks, giving the annotated expression $a \mathrel{⦂} [X]\texttt{GIVEN}\ \mathbb{A}, \texttt{GIVEN}\ \mathbb{A}$. Comparison of the generic type with the instantiated type does not determine a type for the generic parameter X, and so this specification is not well-typed.

Cases where these types are not unique (contain unconstrained variables) are not well-typed.

EXAMPLE 2 The paragraph

$empty == \varnothing$

will contain the annotated expression $\varnothing \mathrel{⦂} [X]\,\mathbb{P}\,X, \mathbb{P}\,\alpha$, in which the type determined for the generic parameter X is unconstrained, and so this specification is not well-typed.

The value of the reference expression that refers to a non-generic definition i is the value of the declaration to which it refers.

$$[\![\, i \,]\!]^{\varepsilon} \;=\; \lambda\ M : Model \bullet M\ i$$

In terms of the semantic universe, its semantic value, given a model M, is that associated with the name i in M.

C.6.29 Generic instantiation

C.6.29.1 Syntax

```
Expression = ...
           | RefName , [-tok , Expression , { ,-tok , Expression } , ]-tok
           | ...
           ;
```

C.6.29.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathrel{⦂} \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \mathrel{⦂} \tau_n}{\Sigma \vdash^{\varepsilon} i[(e_1 \mathrel{⦂} \tau_1), ..., (e_n \mathrel{⦂} \tau_n)] \mathrel{⦂} \tau} \left(\begin{array}{l} i \in dom\ \Sigma \\ generic\_type\ (\Sigma\ i) \\ \tau_1 = \mathbb{P}\,\alpha_1 \\ \quad \vdots \\ \tau_n = \mathbb{P}\,\alpha_n \\ \tau = (\Sigma\ i)\ [\alpha_1, ..., \alpha_n] \end{array} \right)$$

