Z Notation

Final Committee Draft, CD 13568.2

August 24, 1999

Developed by members of the Z Standards Panel

BSI Panel IST/5/-/19/2 (Z Notation)
ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z)
Project number JTC1.22.45

Project editor: Ian Toyn
ian@cs.york.ac.uk
http://www.cs.york.ac.uk/~ian/zstan/

This draft expresses the consensus position of the members of the Z Standards Panel. It is published for review by the wider community: comments should be submitted via national standards bodies. The Foreword includes a comparison with the first committee draft, which will be excluded from the future International Standard.

Contents

Figures

Tables

Foreword

The International Organization for Standardization (ISO) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

This document was prepared by ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z) for project JTC1.22.45. Its structure is in accordance with ISO Directives Part 3. The membership of JTC1/SC22/WG19 (Rapporteur Group for Z) includes the members of BSI Panel IST/5/-/19/2 (Z Notation). Annexes A and B are normative parts of this International Standard; the other annexes are for information only.

The normative clauses of this International Standard organise the language definition as a sequence of phases. This sequence is illustrated in Figure 1 in 5.1, and the corresponding clauses are detailed in 5.3. Informative annex C duplicates some of the definition reorganised by syntactic production, but there is much that does not fit into this organisation, as explained in its introduction.

The following comparison is included only for the benefit of reviewers; it will be excluded from the International Standard.

Comparison with first Committee Draft (Version 1.2) and comments thereon

Many changes since Version 1.2 are identified here, first those with widespread effects through the document, and then those with more localised effects. Attention is drawn to comments that relate to those changes. After that, those comments that are still outstanding are enumerated, and finally the other comments are identified.

Widespread changes

a) The document has been revised to comply with ISO directives (comment UK9).

b) Types have been separated from the semantics (comments UK4–6): elements and situations have gone, and generic lifting has simplified to something that happens within the semantic relation for generic axiomatic description paragraphs. Free variable and alphabet functions have been subsumed by calculations on type signatures. A new meta-operation called *decor* assists in defining schema operations (comment CA10).

c) More consistent metalanguage is now used — for syntactic metalanguage, ISO/IEC 14977:1996 is used (comment CA2), and for semantic metalanguage, a Z-like syntax for ZF set theory is used, with the same precedences as used in Z (comment JP1). Far fewer bracket shapes are used (comments CA18, US33).

d) Many Z notations were not fully defined but now are, e.g. sections (comments JP44, UK1), free types (comment JP49, UK54), numbers (comment UK77), and schema operations (comments JP54, JP80, UK63, US71, US72).

e) The organisation of the normative technical material by syntactic productions has been relegated to an annex, with clauses now organising the normative technical material by phase of definition. This is because some parts of the definition cannot be accommodated by the organisation by syntactic production (proposal IT.50.8).

f) Schema has been merged into Expression (proposal IT.48.2 part c, comments CA36, UK34). The term text or expression is no longer used, as the traditional term schema text now suffices (comment JP4 part 2).

g) Only a few changes have been done to the Z notation itself:

1) free type paragraphs can now define mutually recursive free types;

2) schema piping is included;

3) let and $\exists$ notations with == local declarations are used in place of substitution $\odot$ and $\odot$ notations. If the substitution notation were wanted, it could be defined by the following syntactic transformations.

$$e_{\mathrm{o}} \odot e \implies \mu\ \{e_{\mathrm{o}}\} \bullet e$$
$$e_{\mathrm{o}} \odot p \implies \exists\ \{e_{\mathrm{o}}\} \bullet p$$

Localised changes

a) Acknowledgements of contributors and membership of the Z Standards Panel have been omitted (proposal IT.48.4). There does not seem to be any mention of acknowledgements in ISO directives. I expect my name to disappear from the cover; it is there on this Working Draft so that you know who to send your comments to.

b) The Foreword is where commentary on a particular draft appears, and so is necessarily different from Version 1.2. Two paragraphs of "boilerplate text" provided by the LaTeX macros start the Foreword. They are similar to (though not word-for-word the same as) text present in another standard, though there does not seem to be an ISO directive requiring them (proposal IT.48.1 part c).

c) An alternative Introduction is presented (proposal IT.48.7).

d) A title page has been added, as required by ISO directive 6.1.1 (proposal IT.48.1 part a).

e) The Scope clause has been rewritten in the style required by ISO directive 6.2.1 (proposal IT.48.1 part b).

f) The Conformance clause has been extended. (Comments CA6, CA8 and UK12 noted the need for this.) It has also been moved later in the document, according to ISO directive 5.1.3 (proposal IT.48.3). Discussion of the structure of this standard document has been included here.

g) In Normative references, ISO directive 6.2.2 requires a paragraph of "boilerplate text" at the start, the omission of which in 1.2 has been corrected (proposal IT.48.1 part c). A reference to the ISO standard on which Unicode is based has been added (US13).

h) The three clauses Semantic metalanguage, Semantic universe and Language description have been replaced by the two clauses Terms and definitions, and Symbols and definitions, those being the headings required by ISO for the clauses that introduce the terminology and notation that is to be used in defining the Z notation. The separation of types from the semantics (comments UK4–6) has led to substantial changes in content here. Within Terms and definitions, a style that allows an occurrence of a term to be replaced by its definition in such a way that the sentence in which it appears remains grammatically correct has been used, as required by ISO directive C.1.5.3 (proposal IT.48.1 part b). Within Symbols and definitions, more explanation of the semantic metalanguage has been included (comment US14).

i) Z characters are now defined in terms of Unicode (proposal IT.51.1). The Z characters clause and Mark-ups annex have been separated from Lexis (proposal IT.48.2 part a), arising from progress that allows definition of the widely-used LaTeX mark-up, which was not present in 1.2, and description of how formal and informal parts are delimited (comment US11). The Z characters corresponding to schema outlines are explicitly permitted to be rendered in either of two different ways, corresponding to past practice. UP and DOWN have evolved (comment US146), and digit strokes are specified differently. The syntax is specified in terms of individual strokes, decoration being what is done to a schema by a stroke (comment US40). There are more characters in the SPECIAL class, to reduce the need for spaces between tokens (proposal IT.48.13). The turnstile $\vdash$ has been replaced by $\models$?.

j) Throughout the syntaxes, the ISO standard syntactic metalanguage has been used (comments CA2, US125, US141), and the usual typeset representation of Z symbols is used for corresponding tokens (comments CA3, US5, US122). (This is somewhat of a contradiction, since the current version of the ISO standard syntactic metalanguage does not permit use of mathematical symbols as meta-identifier characters. Worse, reserved characters of the syntactic metalanguage are also used as meta-identifier characters, distinguished by the suffix `-tok` and the use of spaces around every symbol — see 4.1. This latter deviation from the metalanguage standard could be avoided by the use of capitalised names for those tokens, but the use of such capitalised names for all tokens would seriously affect the readability of this standard.) The representation syntax has been merged with the concrete syntax (comments CA17, UK11). Non-terminals are named consistently across different syntaxes (comment JP64).

k) The syntax of paragraphs has been simplified (comment JP58). The operator templates have been simplified somewhat, replacing generic arguments by a separate generic category of operator (comments CA38, JP62). Binding extensions can be empty (comments UK33, US52), and their names shall be distinct (comments CA23, US51). Set extensions can be empty (comments UK30, US44). The syntactic ambiguity concerning logical operators used in schemas that are used as predicates no longer has any semantic significance (comment CA29). Table 27 clarifies where numeric operator precedences interleave with the precedences of syntactic productions (comment CA37). String literals are no longer in the syntax (comments JP4 part 1, UK69, US15, US132). Schema piping expressions have been added (proposal IT.48.14). Semicolon can be used as a conjunction operator between predicates (proposal IT.48.19).

l) The Characterisation rules have been separated into a new clause (proposal IT.50.3). This change in presentation emphasises that these rules effect a separate phase in the definition: they have to be applied exhaustively before the other syntactic transformations, and they are applicable to the concrete syntax, being independent of annotations. Characteristic tuples have been defined by syntactic transformation (comments JP12, US46).

m) The Annotated syntax (renamed from Abstract syntax) has shrunk, as syntactic transformation rules (comments JP55, US123) have been more widely exploited. Given types and generic types have been distinguished, as required by the type inference rules. The effects of operator templates have been formalised using syntactic transformation rules (comment CA40).

n) The Prelude has been introduced, for defining the semantics of number literal expressions.

o) The Type inference rules ban redefinition at top-level (comments CA35, US105). The type rule for schema renaming has been relaxed (comments CA31, CA32, CA33). In type sequents, the syntactic class is distinguished by a superscript on the turnstile instead of :: or $\surd$ (proposal IT.48.9). A notation has been introduced for generic types (proposal IT.48.10).

p) A new clause defines the inference of implicit generic instantiations (comments CA5, US7 and proposal IT.50.4), along with an operation for the substitution of types for generic parameters (proposal IT.48.18). The filling-in of implicit generic instantiations cannot be done until after type inference has been completed, so presenting this phase of processing separately is clearer. Carrier sets have been defined by syntactic transformation, clarifying that they can be empty (comment UK17).

q) The Semantic transformation rules clause has been added. (Hence the other transformations have been renamed as Syntactic transformation rules.) The only available definition of free type paragraphs is a semantic transformation rule. Other semantic transformation rules are exploited to minimise the number of notations that have to be defined by direct relation to ZF set theory.

r) The Semantic relations (renamed from Semantic equations because that for μ is not an equation, Japan [16-1]) are presented in a first-order style (proposal IT.48.2 part d), omitting relations for notations that are now defined by transformations. A little more is said about looseness (comment CA15), the position having changed (proposal SHV.50.1) so that all expressions denote values, though leaving unspecified the values

of undefined definite description and application expressions, and all predicates are either *true* or *false* as determined from the values of their constituent expressions.

s) In Mathematical toolkit, the operator template paragraphs have been rewritten to be consistent with the revised concrete syntax (comments CA45, CA46, UK76). The mathematical toolkit is now based on the 2nd edition of the Z Reference Manual, but with bags excluded (proposal IT.48.2 part e). The domains of some operations have been widened (e.g. comment CA42). The set $\mathbb{A}$ has been added to represent all numbers (comments CA43, UK17, UK82, US157). Definitions for numeric operations have been added (comment UK77), though only over integers (comment UK88), with `div` and `mod` defined as in comment CA44. Negation has been made shorter than subtraction (comment UK85). Tuple selection notation has been exploited (comment CA41). Domain restriction, domain subtraction and extraction have been made right associative (an option that was not available before the introduction of operator templates). As only one name is defined at once, and there are no laws, the sub-headings such as Name, Definition and Description have been omitted, resulting in a presentation more like the other phases of the standard. The mathematical toolkit is presented as a hierarchy of sections, allowing those sections to be reused individually (proposal SHV.51.4).

t) The Interchange format annex has ceased to exist, as no revision of it is available.

u) The Logical theory of Z annex has been omitted (proposal SHV.51.3), as it is not necessary to standardise any particular set of logical inference rules, the semantics being sufficient for verifying the soundness of logical inference rules.

v) The Tutorial annex, illustrating the interpretation of fragments of Z, has been added (comment UK10 and proposal IT.51.3).

w) The Conventions for state-based descriptions annex has been added (comments JP81, UK3, US4).

x) The References annex has been renamed Bibliography (directive 5.2.7), and contains only those items currently referenced.

y) An Index has been added (comments JP82, US2, US3).

Other comments

The following comments on the first CD are those that have not been mentioned above. They either note minor mistakes that are no longer made, have become no longer applicable because of other changes (but were nevertheless valuable at the time), or were rejected: CA{1, 4, 7, 9, 11–14, 16, 19–22, 24–28, 30, 34, 39, 48-62}, JP{2, 3, 5–11, 13–43, 45–48, 50–53, 56, 57, 59–61, 63, 65–79}, UK{2, 7, 8, 13–16, 18–29, 31, 32, 35–53, 55–62, 64–68, 70–75, 78–81, 83, 84, 86, 87, 89–99}, US{1, 6, 8–10, 12, 16–32, 34–39, 41–43, 45, 47–50, 53–70, 73–104, 106–121, 124, 126–131, 133–140, 142–145, 147–156, 158–174}.

Introduction

This International Standard specifies the syntax, type 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.

A specification written in Z 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 a precise syntax and semantics for some 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 to be performed automatically on a specification.

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 software and hardware developments.

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 [3][4]. Early definitions of the notation were made by Sufrin [16] and by King *et al* [9]. Spivey's doctoral thesis showed that the semantics of the notation could be defined in terms of sets of models in ZF set theory [13]. His book "The Z Notation — A Reference Manual" [14][15] is the most complete definition of the notation, prior to this International Standard. This International Standard aims to address issues that have been resolved in different ways by different users, and hence encourage interchange of specifications between diverse tools. It also aims to be a complete formal definition of Z.

Formal Specification — Z Notation — Syntax, Type 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;
- some mark-ups of the Z notation.

The following are outside the scope of this International Standard:

- any method of using Z.

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 14977:1996, *Information Technology — Syntactic Metalanguage — Extended BNF*

ISO 8879:1986(E), *Information Processing — Text and Office Systems — Standard Generalized Mark-up Language (SGML)*

ISO/IEC 10646-1:1993, *Information Technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

3 Terms and definitions

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

3.1
binding
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
conservative extension
extended *theory* such that every *model* of the unextended *theory* is a subset of a *model* for the extended *theory*

3.5
consistent theory
theory whose set of *models* is non-empty

3.6
constraint
property that is either true or false

3.7
environment
function from names to information used in type inference

3.8
inconsistent theory
theory whose set of *models* is empty

3.9
interpretation
function from global names of a section to values in the *semantic universe*

3.10
loose theory
theory whose set of *models* has more than one member

3.11
metalanguage
language used for defining another language

3.12
metavariable
symbol denoting an arbitrary phrase of a particular syntactic class

3.13
model
interpretation that makes the defining *constraints* of the corresponding section be true

3.14
satisfiable section
section having a *consistent theory*

3.15
schema
set of *bindings*

3.16
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

3.17
scope rules
rules determining the *scope of a declaration*

3.18
semantic universe
set of all semantic values, providing representations for both non-generic and generic Z values

3.19
signature
function from names to types

3.20
theorem
conjecture known to be valid

3.21
theory
name of a section and *models* of that section

3.22
type universe
set of all type values, providing representations for all Z types

3.23
uniquely satisfiable section
section whose *theory* has exactly one *model*

3.24
unsatisfiable section
section having an *inconsistent theory*

3.25
ZF set theory
Zermelo-Fraenkel set theory

4 Symbols and definitions

For the purposes of this International Standard, the following definitions of symbols apply.

4.1 Syntactic metalanguage

The syntactic metalanguage used is the subset of the standard ISO/IEC 14977:1996 [8] 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 to be some syntax.
|	separates alternatives.
,	separates notations to be concatenated.
—	separates notation on the left from notation to be excepted on the right.
{ }	bracket notation to be repeated zero or more times.
[ ]	bracket optional notation.
( )	are grouping brackets (parentheses).
' '	encloses terminal symbols.
;	terminates a definition.
(*)*	brackets 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 used infrequently, so is always used with parentheses, making its precedence irrelevant.

EXAMPLE 1 The lexis of a `NUMBER` token, and its informal reading, are as follows.

```
NUMBER              =  DIGIT , { DIGIT } ;
```

The non-terminal symbol `NUMBER` stands for a maximal sequence of one or more 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
                           |  '⊨' | '∀' | '∃' | '•' | '⇔' | '⇒' | '∨' | '∧' | '¬ ' | '∈'
                           |  'λ' | 'μ' | '₉' | '≫' | '|' | '×' | 'ℙ' | 'θ'
                           |  '{' | '}' | '(' | ')' | '[' | ']' | '⦇' | '⦈' | '⟪' | '⟫'
                           |  '|' | ',' | '.' | '/' | ';' | ':' | '=' | '\' | '‗' | '&' | '⨾'
                           ;
?
```

NOTE 1 It is expected that a future version of ISO/IEC 14977:1996 will permit use of Unicode characters, and that a future version of Unicode will permit use of these mathematical symbols.

The new `Meta identifier characters` '(', ')', '[', ']', '{', '}', ',', '|', '&', '; ' and '=' overload existing metalanguage characters. Uses of them as `Meta identifier characters` are with the common suffix -tok, e.g. (-tok, which may be viewed as a postfix metalanguage operator.

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

The syntactic metalanguage is used in defining Z characters, lexis, concrete syntax and annotated syntax.

4.2 Mathematical metalanguage

4.2.1 Introduction

Logic and Zermelo-Fraenkel set theory are the basis for the semantics of Z. In this section the specific notations used are described.

4.2.2 General syntax

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 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 e stands for any proposition or expression. Note that the parentheses containing a Cartesian tuple written with commas may not be elided.

Table 2 – Use of parentheses in metalanguage

Notation	Equivalent to
(e)	e provided e is not of the form $x, y, ..., z$

4.2.3 Propositions

4.2.3.1 Introduction

A Proposition is an expression whose value is either *true* or *false*. The values *true* and *false* are distinct. In a consistent theory a proposition cannot be both *true* and *false*, and the theory in this International Standard is intended to be consistent. Furthermore, every proposition is presumed to be either *true* or *false*, even where it is not possible to say which; that is, the logic is two-valued. In this International Standard it is intended that no use is made of the value of a proposition except where it is possible to establish that value.

4.2.3.2 Propositional connectives

The propositional connectives of negation, conjunction and disjunction are used. In Table 3, P and Q represent arbitrary propositions.

Table 3 – Propositional connectives in metalanguage

Notation	Name	Explanation
$\neg\ P$	negation	*true* iff P is *false*
$P \wedge Q$	conjunction	*true* iff P and Q are both *true*
$P \vee Q$	disjunction	*false* iff P and Q are both *false*

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, $i, ..., k$ are arbitrary identifiers, $S, ..., U$ are arbitrary sets, and $P(i, ..., k)$ and $Q(i, ..., k)$ are arbitrary propositions which may contain references to $i, ..., k$.

Certain abbreviations in the writing of quantifications are permitted, as given in Table 5. They shall be applied repeatedly until none of them is applicable.

Table 4 – Quantifiers in metalanguage

Notation	Name	Explanation
$\exists\ i : S;\ ...;\ k : U \bullet P(i, ..., k)$	existential quantification	there exists values of i in S, ..., k in U such that $P(i, ..., k)$ is true
$\forall\ i : S;\ ...;\ k : U \bullet P(i, ..., k)$	universal quantification	for all i in S, ..., k in U, $P(i, ..., k)$ is true
$\exists_1\ i : S;\ ...;\ k : U \bullet P(i, ..., k)$	unique existential quantification	there exists exactly one configuration of values i in S, ..., k in U such that $P(i, ..., k)$ is true

Table 5 – Abbreviations in quantifications in metalanguage

Notation	Equivalent to
$i, j, ..., k : S$	$i : S;\ j, ..., k : S$
$\exists\ i : S;\ ...;\ k : U \mid P(i, ..., k) \bullet Q(i, ..., k)$	$\exists\ i : S;\ ...;\ k : U \bullet P(i, ..., k) \land Q(i, ..., k)$
$\forall\ i : S;\ ...;\ k : U \mid P(i, ..., k) \bullet Q(i, ..., k)$	$\forall\ i : S;\ ...;\ k : U \bullet (\neg\ P(i, ..., k)) \lor Q(i, ..., k)$
$\exists_1\ i : S;\ ...;\ k : U \mid P(i, ..., k) \bullet Q(i, ..., k)$	$\exists_1\ i : S;\ ...;\ k : U \bullet P(i, ..., k) \land Q(i, ..., k)$

4.2.3.4 Conditional expression and *let*

The conditional expression allows the choice between two alternative values according to the truth or falsehood of a given proposition. Its form is given in Table 6, where P is some proposition and x and y any two expressions. The *let* expression is used to bind the values of variables which are then referred to in the contained clause. Here $i, ..., k$ are identifiers, $x, ..., z$ are expressions and $e(i, ..., k)$ is an expression containing references to $i, ..., k$.

Table 6 – Conditional expression and *let* in metalanguage

Notation	Explanation
if P then x else y	either P is *true* and x is the value, or P is *false* and y is the value
let $i == x;\ ...;\ k == z \bullet e(i, ..., k)$	e with value of x substituted for i, ..., value of z substituted for k

4.2.4 Sets

4.2.4.1 Introduction

The notation used here is based on Zermelo-Fraenkel theory, as described in for example [2], and the presentation here is guided by the order given there. In that theory there are only sets. Members of sets can only be other sets. The word "element" may be used loosely when referring to set members treated as atomic, without regard to their set nature, and abstracting from their representation.

4.2.4.2 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. In Table 7, x is any element (which may be a set) and S is a set.

4.2.4.3 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 trimming back with the axiom of subset or separation.

In the current notation there are corresponding denotations for the empty set, finite set extensions, and set union, and the simple case of comprehensions using the axiom of separation directly. Set intersection and difference are then defined using comprehensions. These are given in Table 8.

Table 7 – Propositions about sets in metalanguage

Notation	Name	Explanation
$x \in S$	membership	*true* iff x is a member of S
$x \notin S$	non-membership	$\neg\ x \in S$
$S \subseteq T$	subset	$\forall\ x : S \bullet x \in T$
$x = y$	equality	for x and y considered as sets, $x \subseteq y \land y \subseteq x$

Table 8 – Set extensions and unions in metalanguage

Notation	Name	Explanation
$\varnothing$	empty set	$x \in \varnothing$ always false
$\{\}$	empty set	$= \varnothing$
$\{x\}$	singleton set	$y \in \{x\}$ iff $y = x$
$S \cup T$	union	the set of x such that $x \in S \lor x \in T$
$\{x, y, ..., z\}$	set extension	$= \{x\} \cup \{y, ..., z\}$
$\{i : S \mid P(i)\}$	comprehension	subset of elements i of S such that $P(i)$, by axiom of separation
$S \cap T$	intersection	$\{x : S \mid x \in T\}$
$S \setminus T$	difference	$\{x : S \mid x \notin T\}$

4.2.4.4 Powersets

The axiom of powers asserts the existence of powersets. The set of all finite subsets is a subset of this. The notation for these two forms is shown in Table 9.

Table 9 – Powerset in metalanguage

Notation	Name	Explanation
$\mathbb{P}\ S$	set of all subsets	$T \in \mathbb{P}\ S$ iff $T \subseteq S$
$\mathbb{F}\ S$	set of all finite subsets	the smallest set containing the empty set and all singleton subsets of S and closed under the operation of forming the union of two sets

4.2.4.5 Numbers

Numbers are not primitive in Zermelo-Fraenkel set theory, but there are several well established ways of representing them. The choice of coding is not specified here. 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, where m and n stand for any expressions whose values are natural numbers.

Table 10 – Operations on numbers in metalanguage

Notation	Explanation
$m + n$	sum of natural numbers m and n
$\#\ S$	cardinality of finite set S
$m\ ..\ n$	set of natural numbers between m and n inclusive

4.2.5 Names

Names are needed for this International Standard. There are several ways of representing them in Zermelo-Fraenkel set theory. The choice of coding is not specified here. Only one operation is needed on names, as defined in Table 11.

Table 11 – Decorations of names in metalanguage

Notation	Explanation
decor $^+$ *i*	name formed by addition of stroke $^+$ to name *i*

4.2.5.1 Tuples and Cartesian products

Tuples and Cartesian products are not primitive in Zermelo-Fraenkel 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 [2]. The choice of coding is not specified here. In this International Standard, particular variables are always known either to have tuples or Cartesian products as their values, or not to have such values. Therefore there is never any possibility of accidental confusion between the encoding used to represent the tuple and any other value which is not a tuple.

In Z, tuples and Cartesian products may have two or more components. In this mathematical language, however, only pairs, which may be iterated, are used.

The syntactic forms are given in Table 12, where p is any pair, $x, ..., z$ are any expressions, and $S, ..., U$ are any sets.

Table 12 – Tuples and Cartesian products in metalanguage

Notation	Explanation
(x, y)	pair
$x \mapsto y$	using "maplet", identical to (x, y)
first p	$\mathit{first}(x, y) = x$
second p	$\mathit{second}(x, y) = y$
$S \times T$	Cartesian product, set of ordered pairs whose first element is in S and whose second element is in T
$(x, y, ..., z)$	abbreviates $(x, (y, ..., z))$
$S \times T \times ... \times U$	abbreviates $S \times (T \times ... \times U)$

4.2.5.2 Further comprehensions and λ

There are notations for more flexible forms of comprehension, whose relationship with the simple form are explained with rewriting rules given in Table 13. In these rules, w is any identifier distinct from those already in use. The rules also refer to a set W, which is required to contain all possible values of the result. In Z itself, the type is used here. The use of mathematical language in this International Standard conforms to an informal type system, as a result of which it is always possible to find a containing set for the value of any comprehension.

This table also explains the notation for λ, which is a form of comprehension convenient when defining functions.

4.2.5.3 Relations

A relation is defined to be a set of Cartesian pairs. There are several operations involving relations, which are given equivalences in Table 14, where Q and R are any relations, and S any set. A proposition about relations is given in Table 15.

4.2.5.4 Functions

A function is identified with a particular form of relation, where each domain element has only one corresponding range element. The phrase "partial function" means exactly the same as the word "function" without qualification. Table 16 shows the various forms of function that are identified. In the table S and T are any sets, and the resultant expressions are sets of functions of various sorts.

Table 13 – Further comprehensions in metalanguage

Notation	Equivalent to
$\{i : S;\ ...;\ k : U \bullet e(i, ..., k)\}$	$\{w : W \mid \exists\ i : S;\ ...;\ k : U \bullet w = e(i, ..., k)\}$ where W is any set containing all values of $e(i, ..., k)$
$\{i : S;\ ...;\ k : U \mid P(i, ..., k) \bullet e(i, ..., k)\}$	$\{w : W \mid \exists\ i : S;\ ...;\ k : U \bullet P(i, ..., k) \land w = e(i, ..., k)\}$ where W is any set containing all values of $e(i, ..., k)$
$\lambda\ i : S \bullet e(i)$	$\{i : S \bullet i \mapsto e(i)\}$
$\lambda\ i : S \mid P(i) \bullet e(i)$	$\{i : S \mid P(i) \bullet i \mapsto e(i)\}$
$\lambda\ i : S;\ ...;\ k : U \bullet e(i, ..., k)$	$\{i : S;\ ...;\ k : U \bullet (i, ..., k) \mapsto e(i, ..., k)\}$
$\lambda\ i : S;\ ...;\ k : U \mid P(i, ..., k) \bullet e(i, ..., k)$	$\{i : S;\ ...;\ k : U \mid P(i, ..., k) \bullet (i, ..., k) \mapsto e(i, ..., k)\}$

Table 14 – Relations in metalanguage

Notation	Name	Definition
$id\ S$	identity function	$\lambda\ x : S \bullet x$
$dom\ R$	domain	$\{p : R \bullet first\ p\}$
$R^\sim$	relational inversion	$\{p : R \bullet second\ p \mapsto first\ p\}$
$S \lhd R$	domain restriction	$\{p : R \mid first\ p \in S\}$
$S \lhd\!\!\!\!- R$	domain subtraction	$\{p : R \mid first\ p \notin S\}$
$R(\!\mid S \mid\!)$	relational image	$\{p : R \mid first\ p \in S \bullet second\ p\}$
$Q \fatsemi R$	relational composition	$\{q : Q;\ r : R \mid second\ q = first\ r \bullet first\ q \mapsto second\ r\}$
$Q \oplus R$	relational overriding	$((dom\ R) \lhd\!\!\!\!- Q) \cup R$

Table 15 – Proposition about relations in metalanguage

Notation	Name	Definition
$Q \approx R$	compatible relations	$(dom\ R) \lhd Q = (dom\ Q) \lhd R$

Table 16 – Functions in metalanguage

Notation	Name	Definition
$S \pfun T$	partial functions	$\{f : \mathbb{P}(S \times T) \mid \forall\ p, q : f \mid first\ p = first\ q \bullet second\ p = second\ q\}$
$S \fun T$	total functions	$\{f : S \pfun T \mid dom\ f = S\}$
$S \rightarrowtail T$	bijections	$\{f : S \fun T \mid f^\sim \in T \fun S\}$
$S \finfun T$	finite functions	$\{f : \mathbb{F}(S \times T) \mid f \in S \pfun T\}$

4.2.5.5 Function use

If f is a function and x is a value in its domain, $f\ x$ represent the corresponding range value, as shown in Table 17. In any case where the value referred to in the table does not exist uniquely, the result of the application is some unknown value.

Table 17 – Function use in metalanguage

Notation	Name	Explanation
$f\ x$	application	the unique value y such that $x \mapsto y \in f$

The mathematical metalanguage is used in type inference rules and in semantic relations.

4.3 Transformation metalanguage

Each transformation rule is written in the following form.

$$\textit{concrete phrase template} \implies \textit{less concrete phrase template}$$

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. 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. Transformations map parse trees of phrases to other parse trees. The metavariables are defined in Tables 18 and 19 (the corresponding syntactic phrases being defined in the syntaxes).

Table 18 – Metavariables for phrases

Symbol	Definition
b	denotes a sequence of digits within a `NUMBER` token.
c	denotes a digit within a `NUMBER` token.
d	denotes a `Paragraph` phrase (d for definition/description).
de	denotes a `Declaration` phrase.
e	denotes an `Expression` phrase.
f	denotes a free type `NAME` token.
g	denotes an injection `NAME` token (g for in$g$ection).
h	denotes an element `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.
se	denotes an `ExpressionList` phrase (se for sequence of expressions).
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.

Table 19 – 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.

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.

$\bowtie$ is used as a reserved stroke, for the generation of new names that are consequently guaranteed not to result in any captures.

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.

The transformation rule metalanguage is used in defining characterisation rules, syntactic transformation rules, type inference rules, instantiation, and semantic transformation rules.

4.4 Type inference rule metalanguage

Each type inference rule is written in the following form.

$$\frac{\mathit{type\ subsequents}}{\mathit{type\ sequent}}(\mathit{side\text{–}condition})$$

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 establishes notation for writing types as `Type` phrases and for writing signatures as `Signature` phrases. The ∈ 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 its carrier set is in the semantic universe.

Type values are formed from just finite sets and ordered pairs, so the mathematical metalanguage introduced in section 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 20, and the names used to denote specific environments are defined in Table 21.

Table 20 – 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 variable names with the name of the ancestral section that originally declared the variable paired with its type.
`SectEnv`	denotes section environments, where `SectEnv` == `NAME` ⇻ `SectTypeEnv`. Section environments associate section names with section-type environments. They are analogous to theories, but with (section name, type) pairs in place of values.

Table 21 – Variables over 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`.

Type sequents are written using the traditional ⊢ notation, but superscripted with a mnemonic letter to distinguish the syntax of the phrase appearing to its right — see Table 22.

NOTE 1 These superscripts are the same as the superscripts used on the ⟦ ⟧ semantic brackets in the semantic relations below.

The annotated phrases to the right of ⊢ in type sequents are phrase templates written using the same metavariables as the syntactic transformation rules; see Table 18.

EXAMPLE 1 The type inference rule for a schema conjunction expression, and its informal reading, are as follows.

Table 22 – Type sequents

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

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathrel{:} \mathbb{P}[\sigma_1] \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathrel{:} \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} e_1 \wedge e_2 \mathrel{:} \mathbb{P}[\sigma_1 \cup \sigma_2]} (\sigma_1 \approx \sigma_2)$$

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.

NOTE 2 The metavariables σ_1 and σ_2 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. $[\![[\![\sigma_1]\!] \cup [\![\sigma_2]\!]]\!]$.

This metalanguage is used in defining type inference rules.

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 — see Table 18.

NOTE 1 The *semantics* parts of semantic relations use the mathematical notation introduced in section 4.2 to manipulate both type values and semantic values. Every Z expression has a semantic value, but the semantic relations do not specify the value to use for undefined expressions. This is why the mathematical metalanguage operations produce values rather than failing when applied to inappropriate arguments.

Symbols concerned with the domain of the semantic definitions are listed in Tables 23 and 24.

The formation of a suitable $\mathbb{W}$ comprising models of sets, tuples and bindings, as needed to model Z, is well-known in ZF set theory and is assumed in this International Standard.

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 — see Table 25.

NOTE 2 The superscripts are the same as those used on the $\vdash$ of type sequents in the type inference rules above.

♡ and ♠ are reserved strokes used in defining the semantics of given types and generic types respectively.

Table 23 – Semantic universe

Symbol	Definition
$\mathbb{W}$	denotes a world of sets, providing semantic values for Z expressions. $\mathbb{W}$ has to be big enough: it shall contain the set NAME from which Z names are drawn and an infinite set and be closed under formation of powersets and products.
$\mathbb{U}$	denotes the semantic universe, providing semantic values for all Z values, where $\mathbb{U} == \mathbb{W} \cup (\mathbb{W} \to \mathbb{W})$. $\mathbb{U}$ comprises $\mathbb{W}$ and Z generic definitions each as a function from the semantic values of its instantiating expressions (a tuple in $\mathbb{W}$) to a member of $\mathbb{W}$.
Model	denotes models, where $Model == \mathtt{NAME} \pfun \mathbb{U}$. Models associate variable names with semantic values. They are applied only to names in their domains, as guaranteed by well-typedness.
Theory	denotes named theories, where $Theory == \mathtt{NAME} \pfun \mathbb{P}\, Model$. Theories associate section names with sets of models.

Table 24 – Variables over semantic universe

Symbol	Definition
M	denotes a model, $M : Model$.
T	denotes a named theory, $T : Theory$.
t	denotes a binding semantic value, $t : \mathtt{NAME} \pfun \mathbb{W}$.
g	denotes a generic semantic value, $g : \mathbb{W} \to \mathbb{W}$.
w, x, y	denote non-generic semantic values, $w : \mathbb{W}$; $x : \mathbb{W}$; $y : \mathbb{W}$.

EXAMPLE 1 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 26 occur repeatedly.

Semantic relation metalanguage is used in defining semantic relations.

Table 25 – Semantic relations

Symbol	Definition
$[\![z]\!]^{\mathcal{Z}}$	denotes the meaning of specification z, where $[\![z]\!]^{\mathcal{Z}} \in \mathit{Theory}$. The meaning of a specification is the set of named theories established by its sections.
$[\![s]\!]^{\mathcal{S}}$	denotes the meaning of section s, where $[\![s]\!]^{\mathcal{S}} \in \mathit{Theory} \pfun \mathit{Theory}$. The meaning of a section is the function from a set of named theories to the set that is the same apart from having an additional member for the given section.
$[\![d]\!]^{\mathcal{D}}$	denotes the meaning of paragraph d, where $[\![d]\!]^{\mathcal{D}} \in \mathit{Model} \rel \mathit{Model}$. 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 $[\![p]\!]^{\mathcal{P}} \in \mathbb{P}\, \mathit{Model}$. 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 $[\![e]\!]^{\mathcal{E}} \in \mathit{Model} \fun \mathbb{W}$. 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 $[\![\tau]\!]^{\mathcal{T}} \in \mathit{Model} \pfun \mathbb{P}\, \mathbb{U}$. The meaning of a type is the semantic value of its carrier set, as determined from the given model.

Table 26 – 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 variables

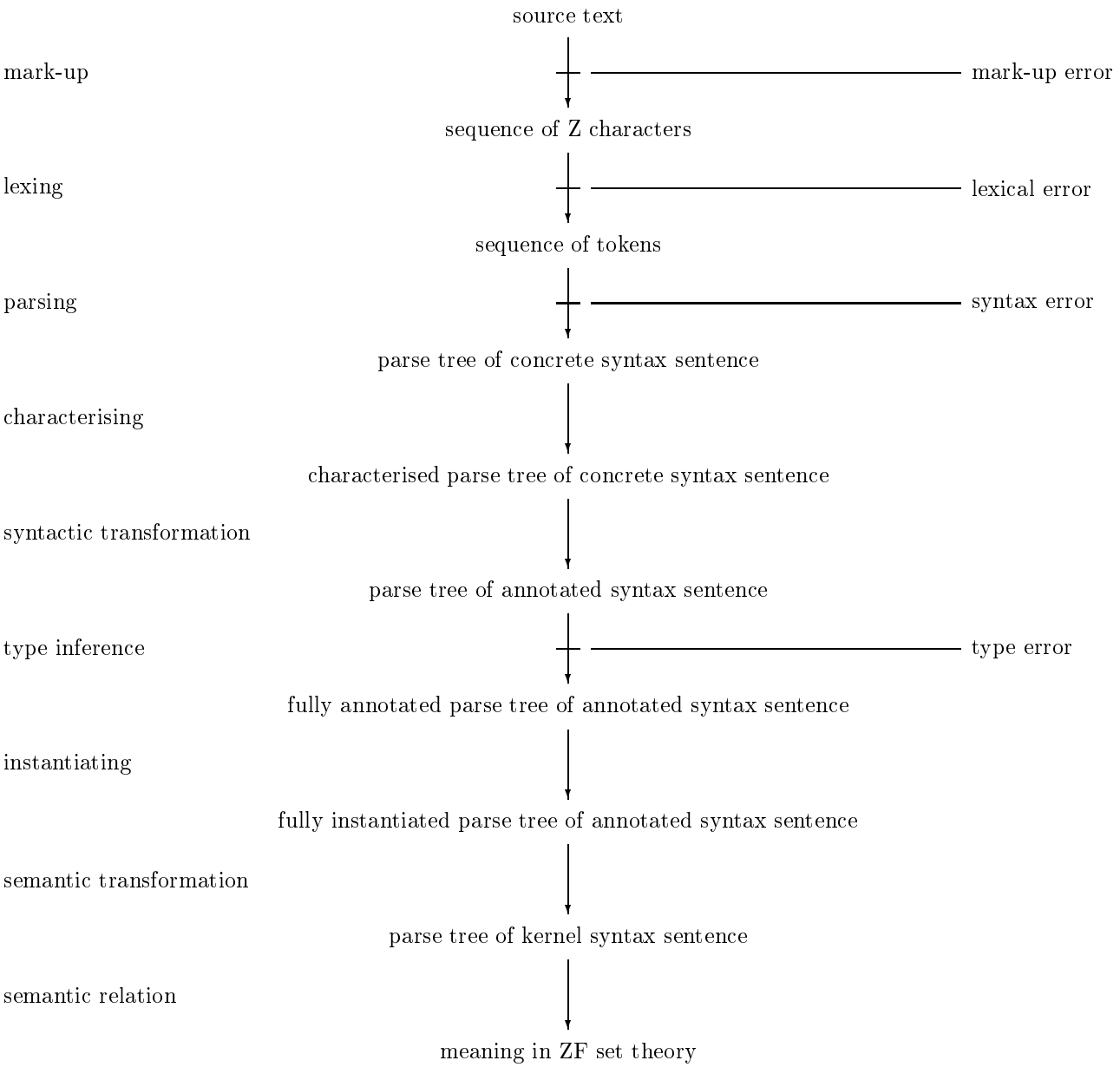
5 Conformance

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

Figure 1 – Phases of the definition



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 which is well-typed according to the type inference system.

NOTE 1 A conforming Z specification may have an inconsistent theory, and consequently be unsatisfiable.

5.2.2 Mark-up conformance

A mark-up for Z based on LaTeX 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 email communications conforms to this International Standard if and only if it follows the rules given for email mark-up in A.3.

NOTE 1 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 that are conforming Z concrete sentences.

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 premisses is contained in the model of the conclusion, i.e. for any rule of inference where p is deduced from p_1, ... p_n,

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

The semantic relation is defined loosely in this International Standard, so as to permit alternative treatments of undefinedness. A Z deductive system may take a particular position on undefinedness. That position should be clearly documented.

5.2.4 Mathematical toolkit conformance

A mathematical toolkit conforms to this International Standard if and only if it defines the same theories as the mathematical toolkit in annex B.

NOTE 1 This permits different formulations of the same definitions.

A Z section whose name is the same as a section of the mathematical toolkit conforms to this International Standard if and only if it defines the same set of models as that section of the mathematical toolkit.

NOTE 2 Alternative and additional toolkits are not precluded, but are required to have different section names to avoid confusion.

5.2.5 Support tool conformance

A strongly conforming Z support tool shall accept all conforming Z specifications and reject all non-conforming Z specifications. 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 1 Strong conformance can be summarised as always being right, whereas weak conformance is never being wrong.

EXAMPLE 1 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.

EXAMPLE 2 The following Z specification conforms to this standard, but a tool might not be able to find the unique assignment of type annotations.

$$S[X] == X$$

$$s == S \vdash \{s \mid x = y\} = \{\} \land s = [x, y, z : \mathbb{A}]$$

EXAMPLE 3 This next example is a trickier variation on Example 2.

$$S[X] == X$$

$$s == S,\ t == S \vdash s = [x, y : \mathbb{A};\ z : \{t \mid x = y\}] \land t = [x, y : \mathbb{A} \mid \theta(s \setminus (z)).x = x]$$

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 representation in Unicode.

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. Some information from parsing operator template paragraphs is fed back to the lexis phase.

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 an initial environment of definitions (which is not apparent in Figure 1).

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.

Clause 14 specifies the instantiation phase, during which reference expressions that refer to generic definitions are translated to generic instantiation expressions.

Clause 15 specifies the semantic transformation phase, during which some annotated parse trees are translated to equivalent other annotated parse trees of a kernel syntax.

Clause 16 specifies the semantic relation between a sentence of the kernel 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 this International Standard; 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 both by their appearances and by relation to 16-bit Unicode [7].

Many Z characters are not present in the standard 7-bit ASCII encoding [5]. 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. Other encodings of Z characters can similarly be defined by relation to the Unicode representation.

6.2 Formal definition of Z characters

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

DIGIT          = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
               | any other Unicode character with the 'decimal' property (as supported)
               ;

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

STROKECHAR     = '′' | '!' | '?' ;

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

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

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

SYMBOL         = '|' | '&' | '⊨' | '∧' | '∨' | '⇒' | '⇔' | '¬' | '∀' | '∃'
               | '×' | '/' | '=' | '∈' | ':' | ';' | ',' | '.' | '•'
               | '\' | '⌈' | '₉' | '≫' | '+'
               | any characters of the mathematical toolkit not included above (as supported)
               | any other Unicode characters not included above (as supported)
               ;
```

6.3 Additional restrictions and notes

The characters enumerated in the formal definition are those used by the core language; they shall be supported. If the mathematical toolkit is supported, then its characters shall be supported. The "other Unicode characters" may also be supported, extending DIGIT, LETTER or SYMBOL according to their property, but not extending SPECIAL.

Use of characters that are absent from Unicode is permitted, but there is no standard way of distinguishing which of DIGIT, LETTER or SYMBOL (not SPECIAL) they extend, and specifications using them might not be interchangeable between tools.

NOTE 1 STROKECHAR characters are used in STROKE tokens in the lexis.

NOTE 2 WORDGLUE characters are used in building NAME tokens in the lexis.

NOTE 3 BOXCHAR characters correspond to Z's distinctive boxes around paragraphs.

NOTE 4 SPACE is a Z character that serves to separate two sequences of Z characters that would otherwise be mis-lexed as a single token.

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).

Unicode: The Unicode value of the Z character.

Spoken name: A suggested form for reading the character 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 1 Not all Z characters are currently supported by Unicode; the exceptions are shown as U+xxxx, and are included in the AMS/STIX mathematical character proposal for the next version of Unicode.

6.4.2 Digit characters

Math	Unicode	Spoken name
0	U+0030	zero | nought
⋮	⋮	⋮
9	U+0039	nine

6.4.3 Letter characters

6.4.3.1 Latin alphabet characters

Math	Unicode	Spoken name
A	U+0041	capital a
⋮	⋮	⋮
Z	U+005A	capital z
a	U+0061	[small] a
⋮	⋮	⋮
z	U+007A	[small] z

6.4.3.2 Greek alphabet characters

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

Math	Unicode	Spoken name
Δ	U+0394	capital delta
Ξ	U+039E	capital xi
θ	U+03B8	[small] theta
λ	U+03BB	[small] lambda
μ	U+03BC	[small] mu

6.4.3.3 Other Z core language letter characters

The other Z core language characters with the Unicode 'letter' property are listed here. (These characters are introduced in the prelude, clause 11.)

Math	Unicode	Spoken name
$\mathbb{A}$	U+xxxx	number | arithmos
$\mathbb{N}$	U+xxxx	nat[ural [number]]
$\mathbb{P}$	U+xxxx	power [set]

6.4.4 Special characters

6.4.4.1 Stroke characters

Math	Unicode	Spoken name
$'$	U+0027	dash | prime
!	U+0021	pling | bang | shriek | exclamation mark
?	U+003F	query | question mark

6.4.4.2 Word glue characters

The characters '$\nearrow$', '$\swarrow$', '$\searrow$', and '$\nwarrow$' 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	Unicode	Spoken name
$\nearrow$	U+2197	up
$\swarrow$	U+2199	end up
$\searrow$	U+2198	down
$\nwarrow$	U+2196	end down
_	U+005F	underscore

6.4.4.3 Bracket characters

Math	Unicode	Spoken name
(	U+0028	(left | open) (round bracket | parenthesis)
)	U+0029	(right | close) (round bracket | parenthesis)
[	U+005B	(left | open) square [bracket]
]	U+005D	(right | close) square [bracket]
{	U+007B	(left | open) brace
}	U+007D	(right | close) brace
⦇	U+xxxx	(left | open) binding [bracket]
⦈	U+xxxx	(right | close) binding [bracket]
⟪	U+300A	(left | open) chevron [bracket]
⟫	U+300B	(right | close) chevron [bracket]

6.4.4.4 Box characters

The `ENDCHAR` character is used to mark the end of a `Paragraph`. The `NLCHAR` character is used to mark a hard newline (see section 7.5). The box rendering of the `BOXCHAR` characters is as lines drawn around the Z text (see section 8.5).

Z character	Simple rendering	Unicode	Spoken name
`AXCHAR`	⎸	U+2577	begin axiomatic
`SCHCHAR`	⌜	U+250C	begin schema
`GENCHAR`	=	U+2550	generic
`ENDCHAR`	(new line)	U+2029	paragraph end
`NLCHAR`	(new line)	U+2028	line end

6.4.4.5 `SPACE` character

The Unicode value of the `SPACE` character is U+0020.

6.4.5 Symbol characters except mathematical toolkit characters

Math	Unicode	Spoken name
|	U+007C	bar
&	U+0026	ampersand
⊨	U+22A8	double turnstile
∧	U+2227	[logical] and
∨	U+2228	[logical] or
⇒	U+21D2	implies
⇔	U+21D4	equivalent | if and only if
¬	U+00AC	not
∀	U+2200	for all
∃	U+2203	exists

×	U+00D7	cross
/	U+002F	[forward] slash
=	U+003D	equals
∈	U+2208	member of | in | element of
:	U+003A	colon
;	U+003B	semi[colon]
,	U+002C	comma
.	U+002E	full stop | period | dot
•	U+xxxx	fat dot | spot | bullet
\	U+xxxx	hide
↾	U+xxxx	project
⨾	U+xxxx	[schema] compose
≫	U+xxxx	[schema] pipe
+	U+002B	plus

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.

Math	Unicode	Spoken name
↔	U+2194	relation
→	U+2192	total function
≠	U+2260	not equal
∉	U+2209	not in
∅	U+2205	empty [set]
⊆	U+2286	subset
⊂	U+2282	proper subset
∪	U+222A	[set] union | cup
∩	U+2229	[set] intersection | cap
\	U+005C	set difference
⊖	U+2296	symmetric set difference
⋃	U+22C3	generalised union
⋂	U+22C2	generalised intersection
𝔽	U+xxxx	finite sets
↦	U+21A6	maplet
⨾	U+xxxx	[relational] compose
∘	U+2218	functional compose
◁	U+25C1	domain restrict
▷	U+25B7	range restrict
⩤	U+xxxx	domain subtract
⩥	U+xxxx	range subtract
~	U+007E	inverse
⦇	U+xxxx	(left | open) relational image [bracket]
⦈	U+xxxx	(right | close) relational image [bracket]
⊕	U+2295	[relational] override

$\pfun$	U+xxxx	[partial] function
$\pinj$	U+xxxx	[partial] injection
$\inj$	U+21A3	total injection
$\psurj$	U+xxxx	[partial] surjection
$\surj$	U+21A0	total surjection
$\bij$	U+xx	bijection
$\ffun$	U+xxxx	finite function
$\finj$	U+xxxx	finite injection
$\num$	U+xxxx	integer
-	U+002D	[unary] minus
$-$	U+2212	[binary] minus
$\leq$	U+2264	less than or equal to
$<$	U+003C	less than
$\geq$	U+2265	greater than or equal to
$>$	U+003E	greater than
#	U+0023	cardinality | hash
$\langle$	U+2329	(left | open) sequence [bracket]
$\rangle$	U+232A	(right | close) sequence [bracket]
$\cat$	U+xxxx	concatenate
$\extract$	U+21BF	extract
$\filter$	U+21BE	filter

6.4.7 Renderings of Z characters

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

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 still represent the same Z character. For example, 'd', 'd', '**d**' and '**d**' are all the same Z character. Although 'dom', 'dom', '**dom**' and 'do**m**' all represent the same token, tokens shall be rendered consistently throughout a specification, to avoid confusion.

For historical reasons, some different Z characters have similar-looking renderings. In particular:

- schema composition '$\fatsemi$' and the mathematical toolkit character relational composition '$\fatsemi$' are different Z characters;
- schema projection '$\project$' and the mathematical toolkit character filter '$\filter$' are different Z characters;
- schema hiding '$\hide$' and the mathematical toolkit character set minus '$\setminus$' 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.

A lexer produces a stream of tokens for consumption by a parser. In the formal definition below, TOKENSTREAM is the start symbol. The tokens are those non-terminal symbols that are enumerated within the TOKEN rule. The SPACE characters are eliminated from the token stream and not passed to the parser.

7.2 Formal definition of lexis

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

TOKEN           = NAME | NUMBER | STROKE
                | (-tok | )-tok | [-tok | ]-tok | {-tok | }-tok | NL
                | section | parents | true | false | let | if | then | else
                | pre | relation | function | generic | leftassoc | rightassoc
                | : | == | ,-tok | ::= | |-tok | & | \ | / | .
                | ; -tok | _ | , , | =-tok | ⟪ | ⟫ | ⊨? | ∀ | •
                | ∃ | ∃1 | ⇔ | ⇒ | ∨ | ∧ | ¬ | ∈ | ℙ
                | × | ⦇ | ⦈ | λ | μ | θ | ⨟ | ≫
                | PREP | PRE | POSTP | POST | IP | I | LP | L | ELP | EL
                | ERP | ER | SRP | SR | EREP | ERE | SREP | SRE | ES | SS
                | AX | GENAX | SCH | GENSCH | END
                ;

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

NUMBER          = DIGIT , { DIGIT } ;

(-tok           = '(' ;
)-tok           = ')' ;
[-tok           = '[' ;
]-tok           = ']' ;
{-tok           = '{' ;
}-tok           = '}' ;

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

NAME            = WORD , { STROKE } ;

WORD            = WORDPART , { WORDPART }
                | LETTER , ALPHASTR , { WORDPART }
                | SYMBOL , SYMBOLSTR , { WORDPART }
                ;
```

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

7.3 Additional lexical restrictions, notes and examples

The non-terminal symbols that appear to be undefined in the formal definition of lexis are keyword tokens (see 7.4).

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 NUMBER, which would otherwise be lexed as a single WORD; between a NAME and a STROKE (a decoration expression), which would otherwise be lexed as a single NAME; and between two consecutive NUMBERs, which would otherwise be lexed as a single NUMBER.

Some tools may impose restrictions on the forms of some names. For example, mapping section names to filenames might be eased by banning decorations and non-ASCII characters.

For every '↘' character in a WORD, there shall be a paired following '↖' character; for every '↗' character in a WORD, there shall be a paired following '↙' character. These shall occur only in nested pairs.

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

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

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

The concrete syntax allows a RefName, which may be a NAME that includes strokes; it also allows an expression to be decorated with a stroke. When the decorated expression is a NAME, the lexis disambiguates the two cases by using the white space between the NAME and the STROKE: in the absence of any white space, the stroke shall be lexed as part of the NAME; in the presence of white space, the stroke shall be lexed as a decoration on the expression.

EXAMPLE 4 $x!$ is the NAME comprising the WORD 'x' followed by the STROKE '!'.
x ! is the decorated expression comprising the RefName expression 'x' decorated with the STROKE '!'.
$x!$! is the decorated expression comprising the RefName expression '$x!$' decorated with the STROKE '!'.

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

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

A multi-digit last WORDPART enclosed in a ↘ ... ↖ pair is deprecated, because of the visual ambiguity with multiple STROKE subscript digits.

NOTE 1 There is no need for any particular mark-up to follow these rules; this syntax applies only to tokens built from Z characters.

NOTE 2 Although a parser does not need to know the spelling of particular instances of NAME, NUMBER, STROKE, etc tokens, subsequent phases of processing such as typechecking do. This relation is not explicitly formalized here.

7.4 Keyword tokens

7.4.1 Introduction

The keywords of the Z core language conform to the lexis of NAME, but the concrete syntax requires distinct tokens to be generated for them: the tokens of the alphabetic and symbolic keywords can be generated only from the

specified sequences of Z characters, not from any other spellings. The following tables show the keywords of the Z core language in their mathematical representation. The columns give:

Math: 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.

Spoken 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 1 Even where a keyword consists of a single Z character, the spoken names may differ. The spoken name of a character tends to reflect its form; that of the token tends to reflect its function.

7.4.2 Alphabetic keywords

Math	Token	Spoken name
else	else	else
false	false	false
function	function	function
generic	generic	generic
if	if	if
leftassoc	leftassoc	left [associative]
let	let	let
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

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

Math	Token	Spoken name
$\langlebar$	$\langlebar$	(left | open) chevron [bracket]
$\rangle\!\rangle$	$\rangle\!\rangle$	(right | close) chevron [bracket]
$\models?$	$\models?$	conjecture
$\forall$	$\forall$	for all
$\bullet$	$\bullet$	fat dot | 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
$\in$	$\in$	member of | in | element of
$\upharpoonright$	$\upharpoonright$	project
$\times$	$\times$	cross
$(\!\mid$	$(\!\mid$	(left | open) binding [bracket]
$\mid\!)$	$\mid\!)$	(right | close) binding [bracket]
λ	λ	lambda
μ	μ	mu
θ	θ	theta
$\fatsemi$	$\fatsemi$	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.

7.4.4 User-defined operators

Each operator template creates additional keyword-like associations between strings of Z characters (`WORD`s) and appropriate tokens. These associations are disabled at the end of the Z section, and may be restored by a parents phrase. An operator template paragraph cannot introduce a token for a `WORD` that is already in the mapping, because that paragraph cannot be parsed. A parent is not permitted to introduce a token for a `WORD` that already has a token associated with it, unless both tokens were introduced by the same operator template paragraph in a common ancestor section. Hence the set of active associations is always a function.

As could be deduced from the concrete syntax, 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 sequence of non-unary relation
SR	right bracket preceded by sequence 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 sequence of tertiary or higher relation
SRE	last word followed by expression and preceded by sequence of tertiary or higher function or generic
ES	middle word preceded by expression of non-unary operator
SS	middle word preceded by sequence of non-unary operator

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

Math	**Token**
$+$	I

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

Math	**Token**
$(\!\|$	EL
$\|\!)$	ER

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

Math	**Token**
disjoint	PREP

EXAMPLE 4 The operator template paragraph for the $(\langle \_ \rangle)$ operator adds two entries to the mapping.

Math	**Token**
$\langle$	L
$\rangle$	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.

  function generic leftassoc parents relation rightassoc section else then
  ::= |-tok ⟪ ⟫ & ⊨? , , ∧ ∨ ⇒ ⇔ × / =-tok ∈ == : ; -tok ,-tok . • \ ↾ ⨟ ≫
  `I IP EL ELP ERE EREP ES SS SRE SREP`

  NOTE 1 All newlines are soft outside a `DeclPart` or a `Predicate`. So tokens that cannot appear in these contexts are in category BOTH. This also includes the box tokens.

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

  if let pre
  [−tok _ ¬ ∀ ∃ $\exists_1$ (−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 NUMBER 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 section 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 `NLCHAR`s 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; there are no terminal symbols within this syntax, so as not to impose any particular mark-up of the notation. 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. The other non-terminal symbols are written in `MixedCase` and are defined within the concrete syntax.

NOTE 1 This concrete syntax aims to define the language clearly to the human reader; it is not necessarily a grammar suitable for processing by machines, as a parser based on these rules would have to look-ahead many tokens to decide which productions to use. A grammar for a larger language that requires only a single token of look-ahead (given a lexical analyser that itself looks-ahead to distinguish the commas in the set extension $\{a,\ b,\ c\}$ from the commas in the set comprehension $\{a,\ b,\ c : t\}$, and the commas in the generic instantion $e\,[a,\ b,\ c]$ from the commas in the application to a schema $e\,[a,\ b,\ c : t]$) has been written and found to be unambiguous by the parser-generator *yacc*. The extra sentences recognised by a parser using that grammar can be eliminated by postprocessing.

8.2 Formal definition of concrete syntax

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

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

Paragraph      = [-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 *)
               | DeclName , == , Expression , END                     (* horizontal definition *)
               | NAME , [-tok , Formals , ]-tok , == , Expression , END
                                                                      (* generic horizontal definition *)
               | GenName , == , Expression , END                      (* generic operator definition *)
               | Freetype , { & , Freetype } , END                    (* free type *)
               | |=? , Predicate , END                                (* conjecture *)
               | [-tok , Formals , ]-tok , |=? , Predicate , END      (* generic conjecture *)
               | 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 *)
           | Application                                        (* function or 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 , . , NUMBER                            (* tuple selection *)
           | θ , Expression , { STROKE }                        (* binding construction *)
           | RefName                                            (* reference *)
           | RefName , [-tok , Expression , { ,-tok , Expression } , ]-tok
                                                                (* generic instantiation *)
           | NUMBER                                             (* 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 = Prec , PrefixTemplate
                 | Prec , PostfixTemplate
                 | Prec , Assoc , InfixTemplate
                 | NofixTemplate
                 ;

Prec            = NUMBER ;

Assoc           = leftassoc
                | rightassoc
                ;

Template        = PrefixTemplate
                | PostfixTemplate
                | InfixTemplate
                | NofixTemplate
                ;

PrefixTemplate  = (-tok , NAME , { ( _ | ,, ) , NAME } , _ , )-tok ;

PostfixTemplate = (-tok , _ , NAME , { ( _ | ,, ) , NAME } , )-tok ;

InfixTemplate   = (-tok , _ , NAME , { ( _ | ,, ) , NAME } , _ , )-tok ;

NofixTemplate   = (-tok , NAME , { ( _ | ,, ) , NAME } , )-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 Operator precedences and associativities

Table 27 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 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 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.

8.4 Additional syntactic restrictions and notes

Operator template paragraphs shall not give different associativities to operators at the same precedence.

`STROKE` is 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 6.2. Other `STROKE`s are considered to be parts of binding construction expressions if they can be. The schema decoration expression interpretation is considered last.

The names within an operator name, i.e. all the `NAME`s in the `Template` rule's four auxiliaries, shall not have any

Table 27 – Operator precedences and associativities

Productions	Associativity
binding construction	
binding selection, tuple selection	
schema renaming	
schema decoration	
application	left
Cartesian product, function or generic operator application	
schema precondition	
schema projection	left
schema hiding	
schema piping	
schema composition	
conditional	
schema negation	
schema conjunction	
schema disjunction	
schema implication	right
schema equivalence	
substitution expression	
definite description	
function construction	
schema quantifications	
relation operator application	
negation	
conjunction	
disjunction	
implication	right
equivalence	
predicate quantifications	
newline conjunction, semicolon conjunction	

`STROKE`s. This is so that any common decoration of those words can be treated as an application of a decorated instance of that operator. Indeed, in an `OpName`, any `STROKE`s shall be the same on every component `NAME`.

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 as a predicate, or the whole logical operation is a predicate involving expressions each used as a predicate. This ambiguity is benign, as the same meaning is ascribed to both interpretations.

The `NUMBER` in a tuple selection expression is interpreted in decimal (base ten). That the number is in the appropriate range is checked by the relevant type inference rule. Leading zeroes shall be accepted and ignored.

NOTE 1 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 2 One way of parsing nested operator applications at different user-defined levels of precedence and associativity is explained by Lalonde and des Rivieres [10]. Using distinct variants of the operator tokens `PRE` | ... | `SS` for relational operators from those for function and generic operators allows that transformation to avoid dealing with those notations whose precedences lie between the relations and the functions, such as the schema operations.

NOTE 3 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* [9], was equivalent to the relation $e_2 \in e_1$. In Standard Z, juxtaposition is the normal form (canonical representation) of all application expressions, and membership is the normal form of all relational predicates.

NOTE 4 There is no production specifically for powerset. It is introduced as a generic operator in the prelude, as otherwise $\mathbb{P}_1$ in the mathematical toolkit could not be parsed.

NOTE 5 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 6 The token ,-tok serves several purposes: it separates `DeclNames` in a `DeclPart`, bindings in a binding extension expression, expressions in a set extension expression, parents in a section header, names in a formal parameter list, names in a hiding list, expressions in a tuple extension expression, renames in a renaming expression, expressions in an instantiation list, and expressions in a sequence argument. Most of these notations occur in unique distinguishable contexts. There are two exceptions, as mentioned in the note in the introduction. One is a `DeclPart` in a set comprehension expression, for example the set comprehension $\{x, y, z : \mathbb{A}\}$, which is tricky for a parser to distinguish from the set extension expression $\{x, y, z\}$. The other is a generic instantiation expression $i[x, y, z]$, which is tricky to distinguish from the application to a schema construction $i[x, y, z : t]$. One solution to this problem is to try both parses and accept the one that works. Alternatively, a lexer can help a parser by providing it with a token that is distinct from ,-tok in the case of a `DeclPart`. A lexer can distinguish commas in a `DeclPart` by looking ahead over alternating `DeclName` phrases and ,-toks to see if that sequence is ended by a ':' or '==' token. This lookahead also results in the commas between bindings in a binding extension expression being represented by the distinct token.

8.5 Box rendering

There are two different sets of box renderings in widespread use, as illustrated here. Any particular specification should 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. 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.

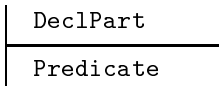
```
[Formals]
DeclPart
---------
Predicate
```

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

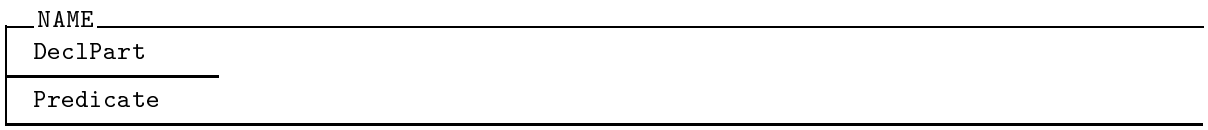
```
NAME [Formals]
DeclPart
---------
Predicate
```

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

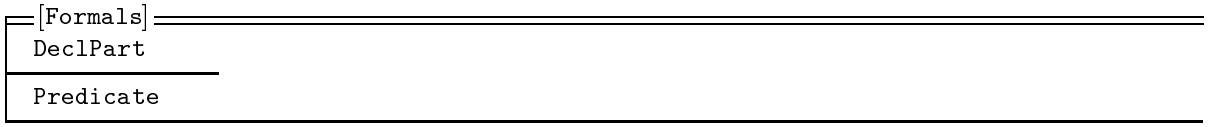
An axiomatic paragraph, involving the AX, |-tok and END tokens, shall have this box rendering.



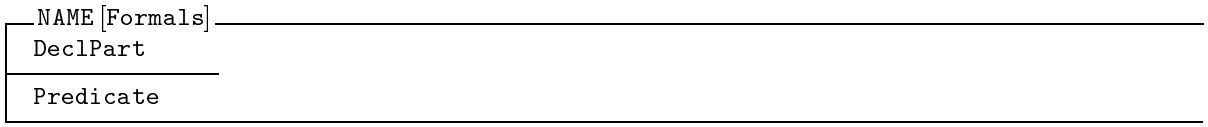
A schema paragraph, involving the SCH, |-tok and END tokens, shall have this box rendering.



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.

Characterisation rules are given for only concrete trees to which they are applicable. 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 auxiliary function *chartuple*.

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.

$$chartuple\ t \implies mktuple\ (charac\ t)$$

$$\begin{aligned}
charac\ (de_1;\ ...;\ de_n \mid p) &\implies charac\ (de_1;\ ...;\ de_n) \\
charac\ (de_1;\ ...;\ de_n) &\implies charac\ de_1 \frown ... \frown charac\ de_n \\
charac\ () &\implies \langle\!|\ |\!\rangle \\
charac\ (i_1, ..., i_n : e) &\implies \langle i_1, ..., i_n \rangle \\
charac\ (i == e) &\implies \langle i \rangle \\
charac\ e &\implies \langle \theta\, e \rangle
\end{aligned}$$

$$\begin{aligned}
mktuple\ \langle e \rangle &\implies e \\
mktuple\ \langle e_1, ..., e_n \rangle &\implies (e_1, ..., e_n)
\end{aligned}$$

NOTE 1 In the last case of *charac*, the type rules ensure that e is a schema.

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.

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

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.

$$\{t\} \implies \{t \bullet chartuple\ t\}$$

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.

$$(\mu\ t) \quad \Longrightarrow \quad \mu\ t \bullet \mathit{chartuple}\ t$$

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

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. Three additional tokens are used besides those defined in the lexis: `GIVEN`, `GENTYPE`, and ⦂. One additional character, ⋈, is included in the `WORDGLUE` class; it is assumed that ⋈ is distinct from the Z characters used in concrete phrases.

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 The annotated syntax is similar to an abstract syntax used in a tool, but the level of abstraction effected by the characterizations and syntactic transformations might not be appropriate for a tool.

10.2 Formal definition of annotated syntax

```
Specification   = { Section } ;

Section         = section , NAME , parents , [ NAME , { ,-tok , NAME } ] , END , { Paragraph } ,
                      [ ⦂ , SectTypeEnv ] ;

Paragraph       = [-tok , NAME , { ,-tok , NAME } , ]-tok , END ,
                      [ ⦂ , [-tok , Signature , ]-tok ]                          (* given types *)
                | AX , Expression , END ,
                      [ ⦂ , [-tok , Signature , ]-tok ]                  (* axiomatic description *)
                | GENAX , [-tok , NAME , { ,-tok , NAME } , ]-tok , Expression , END ,
                      [ ⦂ , [-tok , Signature , ]-tok ]          (* generic axiomatic description *)
                | NAME , ::= , NAME , [ ⟪ , Expression , ⟫ ] ,
                    { |-tok , NAME , [ ⟪ , Expression , ⟫ ] } ,
                    { & , NAME , ::= , NAME , [ ⟪ , Expression , ⟫ ] ,
                    { |-tok , NAME , [ ⟪ , Expression , ⟫ ] } } , END ,
                      [ ⦂ , [-tok , Signature , ]-tok ]                          (* free type *)
                | ⊨? , Predicate , END ,
                      [ ⦂ , [-tok , Signature , ]-tok ]                          (* conjecture *)
                | [-tok , NAME , { ,-tok , NAME } , ]-tok , ⊨? , Predicate , END ,
                      [ ⦂ , [-tok , Signature , ]-tok ]                  (* 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 , . , NUMBER ,
                      [ ⦂ , 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          = GIVEN , NAME                                              (* given type *)
              | GENTYPE , NAME                                            (* generic parameter type *)
              | ℙ , Type                                                  (* set type *)
              | Type , × , Type , { × , Type }                            (* Cartesian product type *)
              | [-tok , Signature , ]-tok                                 (* schema type *)
              | [-tok , NAME , { ,-tok , NAME } , ]-tok , Type , [ ,-tok , Type ]   (* generic type *)
              | τ-tok , { STROKE }                                        (* variable type *)
              ;

Signature     = [ NAME , : , Type , { ; -tok , NAME , : , Type } ] ;
```

10.3 Notes

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

NOTE 2 More types than necessary are permitted by this syntax: generic type notation is used at only the outermost level of a type.

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, and the sequence arguments of operators, via syntactic transformation rules later in this International Standard (see 12.2.5.9). 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.

generic 80 ($\mathbb{P}$ _)

The precedence of the prefix generic operator $\mathbb{P}$ is 80.

$[\mathbb{A}]$

The given type $\mathbb{A}$ 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 12.2.5.9).

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 disjoint 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 1 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).

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

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

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.

Some syntactic transformation rules do not produce annotated parse trees directly, but exhaustive application of the syntactic transformation rules always results in annotated parse trees.

Syntactic transformation rules are given for only concrete trees that are not in the annotated syntax. 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.

12.2 Formal definition of syntactic transformation rules

12.2.1 Specification

12.2.1.1 Anonymous specification

The anonymous specification $d_1 \ldots d_n$ is semantically equivalent to the sectioned specification comprising a single section that has a name — shown here as *Specification* — and whose parents are (implicitly *prelude* and) *standard_toolkit*.

$$d_1 \ldots d_n \implies \text{section } \textit{Specification} \text{ parents } \textit{standard\_toolkit} \ \texttt{END}\ d_1 \ldots d_n$$

The name given to the section is not important: it need not be *Specification*, though it may not be *prelude* or that of any section of the mathematical toolkit.

NOTE 1 If the section is contained in a file, then the name of that file might be a good choice.

12.2.2 Section

12.2.2.1 Base section

The base section section i `END` $d_1 \ldots d_n$ is semantically equivalent to the inheriting section that inherits from no parents (bar *prelude*).

$$\text{section } i \ \texttt{END}\ d_1 \ldots d_n \implies \text{section } i \text{ parents } \texttt{END}\ d_1 \ldots d_n$$

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 $[i_1, \ldots, i_n]$ t `END` can be instantiated to produce a schema definition paragraph.

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

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 $i == e$ `END` introduces the global name i, associating with it the value of e.

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

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 $i\ [i_1, ..., i_n] == e$ `END` can be instantiated to produce a horizontal definition paragraph.

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

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 type paragraph

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

$$... \mid g\langle\langle e \rangle\rangle \mid h \mid ... \quad \Longrightarrow \quad ... \mid h \mid g\langle\langle e \rangle\rangle \mid ...$$

Exhaustive application of this syntactic transformation rule effects a sort.

The second stage requires implicit generic instantiation expressions to have been filled in, which is done during typechecking. Hence that second stage is delayed until after typechecking, where it appears in the form of a semantic transformation rule (section 15.2.3).

12.2.4 Predicate

12.2.4.1 Newline conjunction predicate

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 $p_1 \land p_2$.

12.2.4.2 Semicolon conjunction predicate

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

$$p_1;\ p_2 \quad \Longrightarrow \quad p_1 \land p_2$$

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

12.2.4.3 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.

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

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

12.2.4.4 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*.

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

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

12.2.4.5 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*.

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

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

12.2.4.6 Disjunction predicate

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

$$p_1 \lor p_2 \quad \Longrightarrow \quad \neg\, (\neg\, p_1 \land \neg\, p_2)$$

It is semantically equivalent to not both of them being *false*.

12.2.4.7 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 $\theta\, e$ being a member of the set denoted by schema e.

12.2.4.8 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.

12.2.4.9 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.

12.2.5 Expression

12.2.5.1 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.

12.2.5.2 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.

12.2.5.3 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.

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

It is semantically equivalent to a schema conjunction.

12.2.5.4 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.

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

It is semantically equivalent to a schema disjunction.

12.2.5.5 Schema disjunction expression

The value of the schema disjunction expression $e_1 \lor e_2$ 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.

$$e_1 \lor e_2 \quad \Longrightarrow \quad \neg\, (\neg\, e_1 \land \neg\, e_2)$$

It is semantically equivalent to a schema negation.

12.2.5.6 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*.

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

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.

12.2.5.7 Schema projection expression

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

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

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.

12.2.5.8 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.

12.2.5.9 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 1 These syntactic transformations are applied only to NUMBER tokens that form number literal expressions, not to other NUMBER tokens (those in tuple selection expressions and operator template paragraphs), as those other occurrences of NUMBER do not have semantic values associated with them.

12.2.5.10 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.

12.2.5.11 Parenthesized expression

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

$$(e) \implies e$$

It is semantically equivalent to e.

12.2.6 Schema text

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

12.2.6.1 `Declaration`

Each declaration is transformed to an expression.

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

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

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

$$i_1, ..., i_n : e \implies [i_1 : e \mid \tau_1] \land ... \land [i_n : e \mid \tau_1]$$

12.2.6.2 `DeclPart`

Each declaration part is transformed to an expression.

$$de_1; \ ...; \ de_n \implies de_1 \land ... \land de_n$$

If `NL` tokens have been used in place of any ; s, the same transformation to $\land$ applies.

12.2.6.3 `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 a schema construction expression.

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

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

$$\implies \{\langle\!|\ |\!\rangle\}$$

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.

$$\mid p \implies [\{\langle\!|\ |\!\rangle\} \mid p]$$

12.2.7 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.

12.2.8 Name

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

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

12.2.8.1 `PrefixName`

$$\begin{aligned}
pre\ _ &\implies pre\bowtie \\
prep\ _ &\implies prep\bowtie \\
ln\ _\ ess\ ...\ _\ ess\ _\ ere\ _ &\implies ln\bowtie ess ... \bowtie ess \bowtie ere \bowtie \\
ln\ _\ ess\ ...\ _\ ess\ _\ sre\ _ &\implies ln\bowtie ess ... \bowtie ess \bowtie sre \bowtie \\
lp\ _\ ess\ ...\ _\ ess\ _\ erep\ _ &\implies lp\bowtie ess ... \bowtie ess \bowtie erep \bowtie \\
lp\ _\ ess\ ...\ _\ ess\ _\ srep\ _ &\implies lp\bowtie ess ... \bowtie ess \bowtie srep \bowtie
\end{aligned}$$

12.2.8.2 PostfixName

$$
\begin{array}{rcl}
\_\,post & \Longrightarrow & \bowtie post \\
\_\,postp & \Longrightarrow & \bowtie postp \\
\_\,el\ \_\,ess\ \ldots\ \_\,ess\ \_\,er & \Longrightarrow & \bowtie el \bowtie ess \ldots \bowtie ess \bowtie er \\
\_\,el\ \_\,ess\ \ldots\ \_\,ess\ \_\,sr & \Longrightarrow & \bowtie el \bowtie ess \ldots \bowtie ess \bowtie sr \\
\_\,elp\ \_\,ess\ \ldots\ \_\,ess\ \_\,erp & \Longrightarrow & \bowtie elp \bowtie ess \ldots \bowtie ess \bowtie erp \\
\_\,elp\ \_\,ess\ \ldots\ \_\,ess\ \_\,srp & \Longrightarrow & \bowtie elp \bowtie ess \ldots \bowtie ess \bowtie srp
\end{array}
$$

12.2.8.3 InfixName

$$
\begin{array}{rcl}
\_\,in\,\_ & \Longrightarrow & \bowtie in \bowtie \\
\_\,ip\,\_ & \Longrightarrow & \bowtie ip \bowtie \\
\_\,el\ \_\,ess\ \ldots\ \_\,ess\ \_\,ere\,\_ & \Longrightarrow & \bowtie el \bowtie ess \ldots \bowtie ess \bowtie ere \bowtie \\
\_\,el\ \_\,ess\ \ldots\ \_\,ess\ \_\,sre\,\_ & \Longrightarrow & \bowtie el \bowtie ess \ldots \bowtie ess \bowtie sre \bowtie \\
\_\,elp\ \_\,ess\ \ldots\ \_\,ess\ \_\,erep\,\_ & \Longrightarrow & \bowtie elp \bowtie ess \ldots \bowtie ess \bowtie erep \bowtie \\
\_\,elp\ \_\,ess\ \ldots\ \_\,ess\ \_\,srep\,\_ & \Longrightarrow & \bowtie elp \bowtie ess \ldots \bowtie ess \bowtie srep \bowtie
\end{array}
$$

12.2.8.4 NofixName

$$
\begin{array}{rcl}
ln\ \_\,ess\ \ldots\ \_\,ess\ \_\,er & \Longrightarrow & ln \bowtie ess \ldots \bowtie ess \bowtie er \\
ln\ \_\,ess\ \ldots\ \_\,ess\ \_\,sr & \Longrightarrow & ln \bowtie ess \ldots \bowtie ess \bowtie sr \\
lp\ \_\,ess\ \ldots\ \_\,ess\ \_\,erp & \Longrightarrow & ln \bowtie ess \ldots \bowtie ess \bowtie erp \\
lp\ \_\,ess\ \ldots\ \_\,ess\ \_\,srp & \Longrightarrow & ln \bowtie ess \ldots \bowtie ess \bowtie srp
\end{array}
$$

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.

12.2.9.1 PrefixGenName

$$
\begin{array}{rcl}
pre\ i & \Longrightarrow & pre \bowtie [i] \\
ln\ i_1\ ess\ \ldots\ i_{n-2}\ ess\ i_{n-1}\ ere\ i_n & \Longrightarrow & ln \bowtie ess \ldots \bowtie ess \bowtie ere \bowtie [i_1, \ldots, i_{n-2}, i_{n-1}, i_n] \\
ln\ i_1\ ess\ \ldots\ i_{n-2}\ ess\ i_{n-1}\ sre\ i_n & \Longrightarrow & ln \bowtie ess \ldots \bowtie ess \bowtie sre \bowtie [i_1, \ldots, i_{n-2}, i_{n-1}, i_n]
\end{array}
$$

12.2.9.2 PostfixGenName

$$
i\ post \quad \Longrightarrow \quad \bowtie post\ [i]
$$

$$
\begin{aligned}
i_1 el\ i_2\ ess\ \ldots\ i_{n-1}\ ess\ i_n\ er &\implies \bowtie el \bowtie ess \ldots \bowtie ess \bowtie er\ [i_1, i_2, \ldots, i_{n-1}, i_n] \\
i_1 el\ i_2\ ess\ \ldots\ i_{n-1}\ ess\ i_n\ sr &\implies \bowtie el \bowtie ess \ldots \bowtie ess \bowtie sr\ [i_1, i_2, \ldots, i_{n-1}, i_n]
\end{aligned}
$$

12.2.9.3 `InfixGenName`

$$
\begin{aligned}
i_1 in\ i_2 &\implies \bowtie in \bowtie\ [i_1, i_2] \\
i_1 el\ i_2\ ess\ \ldots\ i_{n-2}\ ess\ i_{n-1}\ ere\ i_n &\implies \bowtie el \bowtie ess \ldots \bowtie ess \bowtie ere \bowtie\ [i_1, i_2, \ldots, i_{n-2}, i_{n-1}, i_n] \\
i_1 el\ i_2\ ess\ \ldots\ i_{n-2}\ ess\ i_{n-1}\ sre\ i_n &\implies \bowtie el \bowtie ess \ldots \bowtie ess \bowtie sre \bowtie\ [i_1, i_2, \ldots, i_{n-2}, i_{n-1}, i_n]
\end{aligned}
$$

12.2.9.4 `NofixGenName`

$$
\begin{aligned}
ln\ i_1\ ess\ \ldots\ i_{n-1}\ ess\ i_n\ er &\implies ln \bowtie ess \ldots \bowtie ess \bowtie er\ [i_1, \ldots, i_{n-1}, i_n] \\
ln\ i_1\ ess\ \ldots\ i_{n-1}\ ess\ i_n\ sr &\implies ln \bowtie ess \ldots \bowtie ess \bowtie sr\ [i_1, \ldots, i_{n-1}, i_n]
\end{aligned}
$$

12.2.10 Relation operator application

All relation operator applications are transformed to annotated membership predicates.

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 rule 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`

$$
\begin{aligned}
prep\ e &\implies e \in prep \bowtie \\
lp\ e_1\ es\ \ldots\ e_{n-2}\ es\ e_{n-1}\ erep\ e_n &\implies (e_1, \ldots, e_{n-2}, e_{n-1}, e_n) \in lp \bowtie es \ldots \bowtie es \bowtie erep \bowtie \\
lp\ e_1\ es\ \ldots\ e_{n-2}\ es\ se_{n-1}\ srep\ e_n &\implies (e_1, \ldots, e_{n-2}, se_{n-1}, e_n) \in lp \bowtie es \ldots \bowtie es \bowtie srep \bowtie
\end{aligned}
$$

12.2.10.2 `PostfixRel`

$$
\begin{aligned}
e\ postp &\implies e \in \bowtie postp \\
e_1\ elp\ e_2\ es\ \ldots\ e_{n-1}\ es\ e_n\ erp &\implies (e_1, e_2, \ldots, e_{n-1}, e_n) \in \bowtie elp \bowtie es \ldots \bowtie es \bowtie erp \\
e_1\ elp\ e_2\ es\ \ldots\ e_{n-1}\ es\ se_n\ srp &\implies (e_1, e_2, \ldots, e_{n-1}, se_n) \in \bowtie elp \bowtie es \ldots \bowtie es \bowtie srp
\end{aligned}
$$

12.2.10.3 `InfixRel`

$$
e_1\ ip_1\ e_2\ ip_2\ e_3\ \ldots \implies e_1\ ip_1\ e_2 \mathbin{:} \tau_1 \wedge e_2 \mathbin{:} \tau_1\ ip_2\ e_3 \mathbin{:} \tau_2\ \ldots
$$

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.

$$\begin{aligned} e_1 = e_2 &\implies e_1 \in \{e_2\} \\ e_1\ ip\ e_2 &\implies (e_1, e_2) \in (\_\ ip\ \_) \end{aligned}$$

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

$$\begin{aligned} e_1\ elp\ e_2\ es\ ...\ e_{n-2}\ es\ e_{n-1}\ erep\ e_n &\implies (e_1, e_2, ..., e_{n-2}, e_{n-1}, e_n) \in \bowtie elp \bowtie es ... \bowtie es \bowtie erep \bowtie \\ e_1\ elp\ e_2\ es\ ...\ e_{n-2}\ es\ se_{n-1}\ srep\ e_n &\implies (e_1, e_2, ..., e_{n-2}, se_{n-1}, e_n) \in \bowtie elp \bowtie es ... \bowtie es \bowtie srep \bowtie \end{aligned}$$

12.2.10.4 `NofixRel`

$$\begin{aligned} lp\ e_1\ es\ ...\ e_{n-1}\ es\ e_n\ erp &\implies (e_1, ..., e_{n-1}, e_n) \in lp \bowtie es ... \bowtie es \bowtie erp \\ lp\ e_1\ es\ ...\ e_{n-1}\ es\ se_n\ srp &\implies (e_1, ..., e_{n-1}, se_n) \in lp \bowtie es ... \bowtie es \bowtie srp \end{aligned}$$

12.2.11 Function or generic operator application

All function operator applications are transformed to annotated application expressions.

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

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 rule 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`

$$\begin{aligned} pre\ e &\implies pre \bowtie e \\ ln\ e_1\ es\ ...\ e_{n-2}\ es\ e_{n-1}\ ere\ e_n &\implies ln \bowtie es ... \bowtie es \bowtie ere \bowtie\ (e_1, ..., e_{n-2}, e_{n-1}, e_n) \\ ln\ e_1\ es\ ...\ e_{n-2}\ es\ se_{n-1}\ sre\ e_n &\implies ln \bowtie es ... \bowtie es \bowtie sre \bowtie\ (e_1, ..., e_{n-2}, se_{n-1}, e_n) \end{aligned}$$

$$\mathbb{P}\ e \implies \mathbb{P}\ e$$

An application of the prefix generic operator $\mathbb{P}$ (that specific `PRE` token) is transformed to a powerset phrase of the annotated notation. Other applications of prefix generic operators are transformed to generic instantiation expressions.

$$\begin{aligned} pre\ e &\implies pre \bowtie\ [e] \\ ln\ e_1\ es\ ...\ e_{n-2}\ es\ e_{n-1}\ ere\ e_n &\implies ln \bowtie es ... \bowtie es \bowtie ere \bowtie\ [e_1, ..., e_{n-2}, e_{n-1}, e_n] \\ ln\ e_1\ es\ ...\ e_{n-2}\ es\ se_{n-1}\ sre\ e_n &\implies ln \bowtie es ... \bowtie es \bowtie sre \bowtie\ [e_1, ..., e_{n-2}, se_{n-1}, e_n] \end{aligned}$$

12.2.11.2 PostfixApp

$$
\begin{array}{rcl}
e\ post & \Longrightarrow & \bowtie post\ e \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-1}\ es\ e_n\ er & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie er\ (e_1, e_2, \ldots, e_{n-1}, e_n) \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-1}\ es\ se_n\ sr & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie sr\ (e_1, e_2, \ldots, e_{n-1}, se_n)
\end{array}
$$

$$
\begin{array}{rcl}
e\ post & \Longrightarrow & \bowtie post\ [e] \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-1}\ es\ e_n\ er & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie er\ [e_1, e_2, \ldots, e_{n-1}, e_n] \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-1}\ es\ se_n\ sr & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie sr\ [e_1, e_2, \ldots, e_{n-1}, se_n]
\end{array}
$$

12.2.11.3 InfixApp

$$
\begin{array}{rcl}
e_1\ in\ e_2 & \Longrightarrow & \bowtie in \bowtie\ (e_1, e_2) \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-2}\ es\ e_{n-1}\ ere\ e_n & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie ere \bowtie\ (e_1, e_2, \ldots, e_{n-2}, e_{n-1}, e_n) \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-2}\ es\ se_{n-1}\ sre\ e_n & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie sre \bowtie\ (e_1, e_2, \ldots, e_{n-2}, se_{n-1}, e_n)
\end{array}
$$

$$
\begin{array}{rcl}
e_1\ in\ e_2 & \Longrightarrow & \bowtie in \bowtie\ [e_1, e_2] \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-2}\ es\ e_{n-1}\ ere\ e_n & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie ere \bowtie\ [e_1, e_2, \ldots, e_{n-2}, e_{n-1}, e_n] \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-2}\ es\ se_{n-1}\ sre\ e_n & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie sre \bowtie\ [e_1, e_2, \ldots, e_{n-2}, se_{n-1}, e_n]
\end{array}
$$

12.2.11.4 NofixApp

$$
\begin{array}{rcl}
ln\ e_1\ es\ \ldots\ e_{n-1}\ es\ e_n\ er & \Longrightarrow & ln \bowtie es \ldots \bowtie es \bowtie er\ (e_1, \ldots, e_{n-1}, e_n) \\
ln\ e_1\ es\ \ldots\ e_{n-1}\ es\ se_n\ sr & \Longrightarrow & ln \bowtie es \ldots \bowtie es \bowtie sr\ (e_1, \ldots, e_{n-1}, se_n)
\end{array}
$$

$$
\begin{array}{rcl}
ln\ e_1\ es\ \ldots\ e_{n-1}\ es\ e_n\ er & \Longrightarrow & ln \bowtie es \ldots \bowtie es \bowtie er\ [e_1, \ldots, e_{n-1}, e_n] \\
ln\ e_1\ es\ \ldots\ e_{n-1}\ es\ se_n\ sr & \Longrightarrow & ln \bowtie es \ldots \bowtie es \bowtie sr\ [e_1, \ldots, e_{n-1}, se_n]
\end{array}
$$

12.2.12 Expression list

$$
e_1, \ldots, e_n \quad \Longrightarrow \quad \{(1, e_1), \ldots, (n, e_n)\}
$$

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

The type inference rules together can be viewed as a partial function that maps a parse tree of an annotated syntax sentence to the unique parse tree of an annotated syntax sentence that is like the given parse tree but with all of the optional annotations present.

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, and allowing well-typedness checks on Z phrases to be automated. 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 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 conjunction of all the type constraints is called the *well-typedness condition*. A sentence is *well-typed* if and only if its well-typedness condition has a solution that provides a unique assignment of type annotations. Sentences that are not well-typed, either because the well-typedness condition has no solution and hence there is no consistent assignment of type annotations, or because the well-typedness condition allows more than one possible assignment of type annotations, are excluded at this stage.

Starting with a Z sentence, the type inference rule for sectioned specification deduces type subsequents for each of the sentence's sections, along with one for the prelude. The type inference rule for inheriting section corresponds to each of those type subsequents. This is the start of a tree of deductions that extends to the atomic phrases of the sentence, namely given types paragraphs, truth predicates, and reference expressions.

The type constraints specified by each deduction are equalities between occurrences of Λ, Γ, Σ, τ and σ variables with formulae on the other side of the deduction, and also the side-conditions of the type inference rules. (Each use of a type inference rule creates new instances of the variables.)

13.2 Formal definition of type inference rules

13.2.1 Specification

13.2.1.1 Sectioned specification

$$\frac{\begin{array}{l}\{\} \vdash^{\mathcal{S}} s_{prelude} \mathbin{⨾} \Gamma_0 \\ \{prelude \mapsto \Gamma_0\} \vdash^{\mathcal{S}} s_{\rho(1)} \mathbin{⨾} \Gamma_{\rho(1)} \\ \ldots \\ \{prelude \mapsto \Gamma_0,\ i_{\rho(1)} \mapsto \Gamma_{\rho(1)},\ \ldots,\ i_{\rho(n-1)} \mapsto \Gamma_{\rho(n-1)}\} \vdash^{\mathcal{S}} s_{\rho(n)} \mathbin{⨾} \Gamma_{\rho(n)}\end{array}}{\vdash^{\mathcal{Z}} s_1 \ldots s_n} (\rho \in 1 \mathinner{..} n \rightarrowtail\!\!\!\!\!\twoheadrightarrow 1 \mathinner{..} n)$$

The sections of a specification can be presented in any order. For a specification to be well-typed, there shall exist a bijection ρ specifying a permutation of the sections so that each section is well-typed in the corresponding section-type environment. The parents relation constrains the permutations that produce a well-typed specification.

The prelude is specified as being included in the environment first. However, when typechecking $\vdash^{\mathcal{Z}} s_{prelude}$, the prelude shall be omitted from the environment.

13.2.2 Section

13.2.2.1 Inheriting section

$$\frac{\Sigma_0 \vdash^{\mathcal{D}} d_1 \mathbin{⨾} [\sigma_1] \quad \ldots \quad \Sigma_{n-1} \vdash^{\mathcal{D}} d_n \mathbin{⨾} [\sigma_n] \quad \Lambda \vdash^{\mathcal{S}} s}{\begin{array}{c}\Lambda \vdash^{\mathcal{S}} \texttt{section}\ i\ \texttt{parents}\ i_1, \ldots, i_m\ \texttt{END} \\ d_1 \mathbin{⨾} [\sigma_1] \ldots d_n \mathbin{⨾} [\sigma_n] \mathbin{⨾} \Gamma\end{array}} \left(\begin{array}{l} i \notin dom\ \Lambda \\ \{i_1, \ldots, i_m\} \subseteq dom\ \Lambda \\ dom\ \sigma_1 \cap dom\ \sigma_2 = \varnothing \land \ldots \land dom\ \sigma_1 \cap dom\ \sigma_n = \varnothing \\ \land \vdots \\ \land\ dom\ \sigma_{n-1} \cap dom\ \sigma_n = \varnothing \\ \Gamma \in (\_ \pfun \_) \end{array} \right)$$

$$
\begin{array}{l}
\textit{where } \Gamma_{-1} = \text{if } i = \textit{prelude} \text{ then } \{\} \text{ else } \Lambda\ \textit{prelude} \\
\quad \textit{and } \Gamma_0 = \Gamma_{-1} \cup \Lambda\ i_1 \cup \ldots \cup \Lambda\ i_m \\
\quad \textit{and } \Gamma = \Gamma_0 \cup \{j : \mathtt{NAME};\ \tau : \mathtt{Type} \mid j \mapsto \tau \in \sigma_1 \cup \ldots \cup \sigma_n \bullet j \mapsto (i,\ \tau)\} \\
\quad \textit{and } \Sigma_0 = \Gamma_0 \fatsemi \textit{second} \\
\quad \textit{and } \Sigma_1 = \Sigma_0 \cup \sigma_1 \ \textit{and} \ \ldots \ \textit{and} \ \Sigma_{n-1} = \Sigma_{n-2} \cup \sigma_{n-1} \\
\quad \textit{and } s = \text{section } i \text{ parents } i_1, \ldots, i_m\ \mathtt{END}\ d_1\ \ldots\ d_{n-1}
\end{array}
$$

Each paragraph of an inheriting section is typechecked in an environment formed from those of the parent sections extended with the signatures of the preceding paragraphs of this section. A further type subsequent checks that the section that remains when the last paragraph is omitted is also well-typed. If the section has no paragraphs, no such type subsequent shall be generated.

NOTE 1 In other words, separate well-typedness conditions are checked for each paragraph-sized prefix of each section. This ensures that the instantiations of references to generics are fully determined before the definition containing those references is used in subsequent paragraphs, and so excludes examples such as the following.

EXAMPLE 1

$$\mid \ empty == \varnothing$$

$$\mid \ inst == empty \cup \{1, 2\}$$

This apparent inefficiency can be avoided in a tool implementation — see 13.3.5.

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

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

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

c) there is no global redefinition between any pair of paragraphs of the section (specified by an enumeration of pairwise disjointness constraints);

d) 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 partial function).

NOTE 2 Ancestors need not be immediate parents, and a section cannot be amongst its own ancestors (no cycles in the parent relation).

NOTE 3 The name of a section can be the same as the name of a declaration — the two are not confused.

13.2.3 Paragraph

13.2.3.1 Given types paragraph

$$\frac{}{\Sigma \vdash^{\mathcal{D}} [i_1, \ldots, i_n]\ \mathtt{END} \fatsemi [i_1 : \mathbb{P}(\mathtt{GIVEN}\ i_1);\ \ldots;\ i_n : \mathbb{P}(\mathtt{GIVEN}\ i_n)]} (\# \{i_1,\ \ldots,\ i_n\} = n)$$

In a given types paragraph, the annotation of the paragraph is a signature associating the given type names with set types. There shall be no duplication of names within a given types paragraph.

13.2.3.2 Axiomatic description paragraph

$$\frac{\Sigma \vdash^{\mathcal{E}} e \fatsemi \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{D}} \mathtt{AX}\ e\ \mathtt{END} \fatsemi [\sigma]}$$

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

13.2.3.3 Generic axiomatic description paragraph

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\mathtt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\mathtt{GENTYPE}\ i_n)\} \vdash^{\mathcal{E}} e \mathbin{:} \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{D}} \mathtt{GENAX}\ [i_1, ..., i_n]\ e\ \mathtt{END} \mathbin{:} [\lambda\ j : dom\ \sigma \bullet [i_1, ..., i_n](\sigma\ j)]} (\#\{i_1,\ ...,\ i_n\} = n)$$

In a generic axiomatic description paragraph $\mathtt{GENAX}\ [i_1, ..., i_n]\ e\ \mathtt{END}$, 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. There shall be no duplication of names within the generic parameters of a generic axiomatic description paragraph.

13.2.3.4 Free type paragraph

$$\frac{\begin{array}{l}\Sigma_o \vdash^{\mathcal{E}} e_{1\,1} \mathbin{:} \mathbb{P}\,\tau_{1\,1} \quad ... \quad \Sigma_o \vdash^{\mathcal{E}} e_{1\,n_1} \mathbin{:} \mathbb{P}\,\tau_{1\,n_1} \\ \vdots \\ \Sigma_o \vdash^{\mathcal{E}} e_{r\,1} \mathbin{:} \mathbb{P}\,\tau_{r\,1} \quad ... \quad \Sigma_o \vdash^{\mathcal{E}} e_{r\,n_r} \mathbin{:} \mathbb{P}\,\tau_{r\,n_r}\end{array}}{\Sigma \vdash^{\mathcal{D}} d \mathbin{:} [\sigma]} \left(\begin{array}{l} \#\{f_1,\ h_{1\,1},\ ...,\ h_{1\,m_1},\ g_{1\,1},\ ...,\ g_{1\,n_1}, \\ \quad \vdots, \\ \quad f_r,\ h_{r\,1},\ ...,\ h_{r\,m_r},\ g_{r\,1},\ ...,\ g_{r\,n_r}\} \\ = r + m_1 + ... + m_r + n_1 + ... + n_r \end{array} \right)$$

$$\begin{array}{lcl} \textit{where}\ \Sigma_o = \Sigma \oplus \{f_1 \mapsto \mathbb{P}\,f_1,\ ...,\ f_r \mapsto \mathbb{P}\,f_r\} \\ \quad \textit{and}\ d & = & f_1 ::= h_{1\,1} \mid ... \mid h_{1\,m_1} \mid g_{1\,1}\langle\!\langle e_{1\,1} \rangle\!\rangle \mid ... \mid g_{1\,n_1}\langle\!\langle e_{1\,n_1} \rangle\!\rangle \\ & & \&\ ...\ \& \\ & & f_r ::= h_{r\,1} \mid ... \mid h_{r\,m_r} \mid g_{r\,1}\langle\!\langle e_{r\,1} \rangle\!\rangle \mid ... \mid g_{r\,n_r}\langle\!\langle e_{r\,n_r} \rangle\!\rangle\ \mathtt{END} \\ \quad \textit{and}\ \sigma & = & f_1 : \mathbb{P}\,f_1;\ h_{1\,1} : f_1;\ ...;\ h_{1\,m_1} : f_1;\ g_{1\,1} : \mathbb{P}(\tau_{1\,1} \times f_1);\ ...;\ g_{1\,n_1} : \mathbb{P}(\tau_{1\,n_1} \times f_1) \\ & & ;\ ...; \\ & & f_r : \mathbb{P}\,f_r;\ h_{r\,1} : f_r;\ ...;\ h_{r\,m_r} : f_r;\ g_{r\,1} : \mathbb{P}(\tau_{r\,1} \times f_r);\ ...;\ g_{r\,n_r} : \mathbb{P}(\tau_{r\,n_r} \times f_r) \end{array}$$

In a free type paragraph d, as expanded in the second local definition, 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 relevant type. There shall be no duplication of names amongst the free types, elements and injections of a free type paragraph.

13.2.3.5 Conjecture paragraph

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \models?\ p\ \mathtt{END} \mathbin{:} [\,]}$$

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

13.2.3.6 Generic conjecture paragraph

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\mathtt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\mathtt{GENTYPE}\ i_n)\} \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} [i_1, ..., i_n] \models?\ p\ \mathtt{END} \mathbin{:} [\,]} (\#\{i_1,\ ...,\ i_n\} = n)$$

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

13.2.4 Predicate

13.2.4.1 Membership predicate

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{:} \tau \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathbin{:} \mathbb{P}\tau}{\Sigma \vdash^{\mathcal{P}} e_1 \in e_2}$$

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.

13.2.4.2 Truth predicate

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

A truth predicate is always well-typed.

13.2.4.3 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.

13.2.4.4 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.

13.2.4.5 Universal quantification predicate

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \forall\, e \bullet p}$$

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.

13.2.4.6 Unique existential quantification predicate

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \exists_1\, e \bullet p}$$

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.

13.2.5 Expression

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

NOTE 2 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.

$$\frac{}{\Sigma \vdash^{\varepsilon} i \mathbin{:} \tau}(i \in dom\ \Sigma)$$

where $\tau =$ if $\Sigma\ i = [i_1, ..., i_n]\tau_0$ then $\Sigma\ i$, $(\Sigma\ i)\ [\tau_1, ..., \tau_n]$ else $\Sigma\ i$

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 variable types (which the context should constrain to non-generic types). Otherwise (if the type in the environment is non-generic), that is the type of the whole expression.

NOTE 3 The operation of generic type instantiation is defined in 14.3.

NOTE 4 If the type is generic, the next phase of processing makes the implicit instantiation explicit, transforming the reference expression to a generic instantiation expression. That cannot be done here, as the new variable types $\tau_1, ..., \tau_n$ have yet to be constrained.

13.2.5.2 Generic instantiation expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \mathbb{P}\tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \mathbin{:} \mathbb{P}\tau_n}{\Sigma \vdash^{\varepsilon} i\ [e_1, ..., e_n] \mathbin{:} (\Sigma\ i)\ [\tau_1, ..., \tau_n]}(i \in dom\ \Sigma)$$

In a generic instantiation expression $i\ [e_1, ..., e_n]$, the expressions shall be sets, and the name i shall be associated with a generic type in the environment. The type of the whole expression is the instantiation of that generic type by the types of those sets.

NOTE 1 The operation of generic type instantiation is defined in 14.3.

13.2.5.3 Set extension expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \tau \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \mathbin{:} \tau}{\Sigma \vdash^{\varepsilon} \{e_1, ..., e_n\} \mathbin{:} \mathbb{P}\tau}$$

In a set extension expression, every component expression shall be of the same type. The type of the whole expression is a set of the components' type.

13.2.5.4 Set comprehension expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\varepsilon} e_2 \mathbin{:} \tau}{\Sigma \vdash^{\varepsilon} \{e_1 \bullet e_2\} \mathbin{:} \mathbb{P}\tau}$$

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 set of the type of expression e_2, as determined in an environment overridden by the signature of schema e_1.

13.2.5.5 Powerset expression

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{:} \mathbb{P}\tau}{\Sigma \vdash^{\varepsilon} \mathbb{P}e \mathbin{:} \mathbb{P}\mathbb{P}\tau}$$

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

13.2.5.6 Tuple extension expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \mathbin{:} \tau_n}{\Sigma \vdash^{\varepsilon} (e_1, ..., e_n) \mathbin{:} \tau_1 \times ... \times \tau_n}$$

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.

13.2.5.7 Tuple selection expression

$$\frac{\Sigma \vdash^{\varepsilon} e \mathrel{:} \tau_1 \times ... \times \tau_n}{\Sigma \vdash^{\varepsilon} e \,.\, b \mathrel{:} \tau_b}(b \in 1 \,..\, n)$$

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

13.2.5.8 Binding extension expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathrel{:} \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \mathrel{:} \tau_n}{\Sigma \vdash^{\varepsilon} \langle\!| i_1 == e_1, ..., i_n == e_n |\!\rangle \mathrel{:} [i_1 : \tau_1;\ ...;\ i_n : \tau_n]}(\#\{i_1,\ ...,\ i_n\} = n)$$

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

13.2.5.9 Binding construction expression

$$\frac{\Sigma \vdash^{\varepsilon} e \mathrel{:} \mathbb{P}[i_1 : \tau_1;\ ...;\ i_n : \tau_n] \qquad \Sigma \vdash^{\varepsilon} i_1{}^* \mathrel{:} \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} i_n{}^* \mathrel{:} \tau_n}{\Sigma \vdash^{\varepsilon} \theta\, e\, {}^* \mathrel{:} [i_1 : \tau_1;\ ...;\ i_n : \tau_n]}$$

In a binding construction expression $\theta\, e\, {}^*$, the expression e shall be a schema, and in the environment shall appear names, like those in the signature of the schema but with the (optional) strokes appended, associated with the same types as those names have in the signature of schema e. The type of the whole expression is that of a binding whose signature is that of the schema.

NOTE 1 The reference expressions $i_1{}^* ... i_n{}^*$ cannot refer to generic declarations, because $\tau_1 ... \tau_n$ are the types of schema components, which cannot be generic types.

13.2.5.10 Binding selection expression

$$\frac{\Sigma \vdash^{\varepsilon} e \mathrel{:} [\sigma]}{\Sigma \vdash^{\varepsilon} e \,.\, i \mathrel{:} \sigma\ i}(i \in dom\ \sigma)$$

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.

13.2.5.11 Application expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathrel{:} \mathbb{P}(\tau_1 \times \tau_2) \qquad \Sigma \vdash^{\varepsilon} e_2 \mathrel{:} \tau_1}{\Sigma \vdash^{\varepsilon} e_1\ e_2 \mathrel{:} \tau_2}$$

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.

13.2.5.12 Definite description expression

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathrel{:} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\varepsilon} e_2 \mathrel{:} \tau}{\Sigma \vdash^{\varepsilon} \mu\ e_1 \bullet e_2 \mathrel{:} \tau}$$

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.

13.2.5.13 Variable construction expression

$$\frac{\Sigma \vdash^{\varepsilon} e \mathrel{:} \mathbb{P}\,\tau}{\Sigma \vdash^{\varepsilon} [i : e] \mathrel{:} \mathbb{P}[i : \tau]}$$

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 set e.

13.2.5.14 Schema construction expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathrel{:} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{E}} [e \mid p] \mathrel{:} \mathbb{P}[\sigma]}$$

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.

13.2.5.15 Schema negation expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathrel{:} \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{E}} \neg\, e \mathrel{:} \mathbb{P}[\sigma]}$$

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.

13.2.5.16 Schema conjunction expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathrel{:} \mathbb{P}[\sigma_1] \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathrel{:} \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} e_1 \wedge e_2 \mathrel{:} \mathbb{P}[\sigma_1 \cup \sigma_2]} (\sigma_1 \approx \sigma_2)$$

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.

13.2.5.17 Schema hiding expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathrel{:} \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{E}} e \setminus (i_1, ..., i_n) \mathrel{:} \mathbb{P}[\{i_1, \ldots, i_n\} \lhd \sigma]} (\{i_1, \ldots, i_m\} \subseteq dom\ \sigma)$$

In a schema hiding expression $e \setminus (i_1, ..., i_n)$, expression e shall be a schema, and the names 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 to be hidden.

13.2.5.18 Schema universal quantification expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathrel{:} \mathbb{P}[\sigma_1] \qquad \Sigma \oplus \sigma_1 \vdash^{\mathcal{E}} e_2 \mathrel{:} \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} \forall\, e_1 \bullet e_2 \mathrel{:} \mathbb{P}[dom\ \sigma_1 \lhd \sigma_2]} (\sigma_1 \approx \sigma_2)$$

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.

13.2.5.19 Schema unique existential quantification expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathrel{:} \mathbb{P}[\sigma_1] \qquad \Sigma \oplus \sigma_1 \vdash^{\mathcal{E}} e_2 \mathrel{:} \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} \exists_1\, e_1 \bullet e_2 \mathrel{:} \mathbb{P}[dom\ \sigma_1 \lhd \sigma_2]} (\sigma_1 \approx \sigma_2)$$

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.

13.2.5.20 Schema renaming expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathrel{:} \mathbb{P}[\sigma_1]}{\Sigma \vdash^{\mathcal{E}} e\ [j_1 / i_1, ..., j_n / i_n] \mathrel{:} \mathbb{P}[\sigma_2]} \left(\begin{array}{l} \#\{i_1, \ldots, i_n\} = n \\ \sigma_2 \in (\_ \nrightarrow \_) \end{array} \right)$$

where $\sigma_2 = (id\ (dom\ \sigma_1) \oplus \{i_1 \mapsto j_1,\ ...,\ i_n \mapsto j_n\})^\sim \fatsemi \sigma_1$

In a schema renaming expression $e\ [j_1\ /\ i_1, ..., j_n\ /\ i_n]$, expression e shall be a schema. There shall be no duplicates amongst the old names $i_1, ..., i_n$. Declarations that are merged by the renaming shall have the same type. 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.

NOTE 1 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.

13.2.5.21 Schema precondition expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \fatsemi \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{E}} \mathrm{pre}\, e \fatsemi \mathbb{P}[\{i, j : \mathtt{NAME} \mid j \in dom\ \sigma \land (j = decor\ '\ i \lor j = decor\ !\ i) \bullet j\} \lhd \sigma]}$$

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.

13.2.5.22 Schema composition expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \fatsemi \mathbb{P}[\sigma_1] \qquad \Sigma \vdash^{\mathcal{E}} e_2 \fatsemi \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} e_1 \fatsemi e_2 \fatsemi \mathbb{P}[\sigma_3 \cup \sigma_4]} \left(\begin{array}{l} \sigma_3 \approx \sigma_4 \\ \{i : match \bullet i \mapsto \sigma_1(decor\ '\ i)\} \approx \{i : match \bullet i \mapsto \sigma_2 i\} \end{array} \right)$$

where $match = \{\ i : dom\ \sigma_2 \mid decor\ '\ i \in dom\ \sigma_1 \land (\forall\ j : \mathtt{NAME} \bullet \neg\ i = decor\ '\ j) \bullet i\}$
and $\sigma_3 = \{i : match \bullet decor\ '\ i\} \lhd \sigma_1$
and $\sigma_4 = match \lhd \sigma_2$

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 unprimed names in schema e_2 for which there are matching primed names in schema e_1. Let σ_3 be the signature formed from the components of e_1 excluding the matched primed components. Let σ_4 be the signature formed from the components of e_2 excluding the matched unprimed components. Signatures σ_3 and σ_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 σ_3 and σ_4.

NOTE 1 This notation would not be associative without the restriction concerning names being unprimed.

13.2.5.23 Schema piping expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \fatsemi \mathbb{P}[\sigma_1] \qquad \Sigma \vdash^{\mathcal{E}} e_2 \fatsemi \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} e_1 \gg e_2 \fatsemi \mathbb{P}[\sigma_3 \cup \sigma_4]} \left(\begin{array}{l} \sigma_3 \approx \sigma_4 \\ \{i : match \bullet i \mapsto \sigma_1(decor\ !\ i)\} \approx \{i : match \bullet i \mapsto \sigma_2(decor\ ?\ i)\} \end{array} \right)$$

where $match = \{\ i : \mathtt{NAME} \mid decor\ !\ i \in dom\ \sigma_1 \land decor\ ?\ i \in dom\ \sigma_2 \land (\forall\ j : \mathtt{NAME} \bullet \neg\ i = decor\ !\ j) \bullet i\}$
and $\sigma_3 = \{i : match \bullet decor\ !\ i\} \lhd \sigma_1$
and $\sigma_4 = \{i : match \bullet decor\ ?\ i\} \lhd \sigma_2$

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 unshrieked names for which there are shrieked names in schema e_1 matching queried names in schema e_2. Let σ_3 be the signature formed from the components of e_1 excluding the matched shrieked components. Let σ_4 be the signature formed from the components of e_2 excluding the matched queried components. Signatures σ_3 and σ_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 σ_3 and σ_4.

NOTE 1 This notation would not be associative without the restriction concerning names being unshrieked.

13.2.5.24 Schema decoration expression

$$\frac{\Sigma \vdash^{\mathcal{E}} e \fatsemi \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{E}} e^{+} \fatsemi \mathbb{P}[\{i : dom\ \sigma \bullet decor^{+}\ i \mapsto \sigma\ i\}]}$$

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 decoration appended to each of its names.

13.3 Notes on properties of the type inference system

13.3.1 Termination

NOTE 1 In every type inference rule bar one, the formula to the right of the turnstile in each of the premiss type sequents above the line appears as a sub-formula of the formula to the right of the turnstile in the type sequent below the line (ignoring any type annotations on those formulae). The exceptional case is the type inference rule for binding construction (θ) expressions, which generates additional premiss type sequents for the optionally decorated names from the operand expression's signature. Type inference on those additional premisses is immediate. The decrease in formula size is thus sufficient to guarantee termination.

13.3.2 Uniqueness

NOTE 1 For any syntactically well-formed Z specification, the one-to-one mapping between annotated syntax productions and type inference rules guarantees that a unique finite type deduction tree of inferences can be generated, along with a unique well-typedness condition comprising a finite number of conjuncts. Only if there is a unique solution to the well-typedness condition is the Z specification well-typed.

13.3.3 Efficiency

NOTE 1 The above arguments for termination and uniqueness show that the cost of applying the type system is linear in the number of formulae in the sentence.

13.3.4 Scope rules

NOTE 1 Here is an informal explanation of the scope rules implied by the above type inference rules.

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 type 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 type 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 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.

13.3.5 Implementation

NOTE 1 The well-typedness condition is likely to be solved by *unification*. Some unification may be done as each deduction generates new type constraints, rather than leaving it until all type constraints are known. The check that any solution is unique can then be implemented by a separate *normalization* phase, which searches for any remaining genericity. Applying that phase to each individual paragraph immediately after typechecking the paragraph efficiently implements the multiple well-typedness conditions noted under the inheriting section type inference rule (13.2.2.1).

Note that the type system is formulated as being applicable only to a whole Z specification sentence, not a smaller phrase. Typechecking smaller phrases in isolation is not a concern in this International Standard, but the only real difficulty is easily identified: all type inference rules except that for a whole specification have an original type sequent below the line in which there is a type environment; so to typecheck a phrase, a suitable type environment shall be provided.

14 Instantiation

14.1 Introduction

The instantiation rule maps each reference expression of generic type to a generic instantiation expression. The transformation is performed on the parse trees of the phrases. Its formal definition requires two auxiliaries: carrier set and generic type instantiation.

14.2 Carrier set

The meta-function *carrier* syntactically transforms a type phrase to an expression phrase denoting the carrier set of that type. As well as being used in this clause for the calculation of implicit generic actuals, it is also used later in semantic transformation rules.

$$
\begin{aligned}
\mathit{carrier}\ (\mathtt{GIVEN}\ i) &\Longrightarrow i \mathrel{⦂} \mathbb{P}(\mathtt{GIVEN}\ i) \\
\mathit{carrier}\ (\mathtt{GENTYPE}\ i) &\Longrightarrow i \mathrel{⦂} \mathbb{P}(\mathtt{GENTYPE}\ i) \\
\mathit{carrier}\ (\mathbb{P}\,\tau) &\Longrightarrow \mathbb{P}(\mathit{carrier}\ \tau) \mathrel{⦂} \mathbb{P}\,\mathbb{P}\,\tau \\
\mathit{carrier}\ (\tau_1 \times ... \times \tau_n) &\Longrightarrow (\mathit{carrier}\ \tau_1 \times ... \times \mathit{carrier}\ \tau_n) \mathrel{⦂} \mathbb{P}(\tau_1 \times ... \times \tau_n) \\
\mathit{carrier}\ ([i_n : \tau_n;\ ...;\ i_n : \tau_n]) &\Longrightarrow [i_n : \mathit{carrier}\ \tau_n;\ ...;\ i_n : \mathit{carrier}\ \tau_n] \mathrel{⦂} \mathbb{P}[i_n : \tau_n;\ ...;\ i_n : \tau_n]
\end{aligned}
$$

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 and the generation of type annotations, each of these transformations generates an expression that has the same textual appearance as the type.

NOTE 3 There is no transformation rule for variable type because in a well-typed specification all variable types have been unified with other types. There is no transformation rule for generic types because they appear in only the type annotation of generic axiomatic paragraphs, and *carrier* is never applied there.

14.3 Generic type instantiation

The meta-function of instantiating a generic type syntactically transforms a generic type and a sequence of argument types to a Z expression denoting the type in which each reference to a generic parameter is substituted with the corresponding argument type. This meta-function is used in two places: in the type inference rule for generic instantiation expressions, where the argument types are all variable types, and in the instantiation rule later in this clause, where the argument types are as already determined by type inference.

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

NOTE 1 There is no transformation rule for variable type because in a well-typed specification all variable types have been unified with other types.

14.4 Formal definition of instantiation rule

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

$$
i \mathrel{⦂} [i_1,\ ...,\ i_n]\tau, \tau' \Longrightarrow i\ [\mathit{carrier}\ \tau_1, ..., \mathit{carrier}\ \tau_n] \mathrel{⦂} \tau'
$$

$$
\mathit{where}\ \exists_1\ \tau_1,\ ...,\ \tau_n : \mathtt{Type} \bullet \tau' = ([i_1,\ ...,\ i_n]\tau)\ [\tau_1,\ ...,\ \tau_n]
$$

It is semantically equivalent to the generic instantiation expression whose generic actuals are the carrier sets of the types inferred for the generic parameters.

NOTE 1 τ' is an instantiation of the generic type appearing as the first component of the pair. The types τ_1, ..., τ_n can be determined by comparison of τ with τ' as suggested by the *where* clause.

15 Semantic transformation rules

15.1 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.

15.2 Formal definition of semantic transformation rules

15.2.1 Specification

There are no semantic transformation rules for specifications.

15.2.2 Section

There are no semantic transformation rules for sections.

15.2.3 Paragraph

15.2.3.1 Free type paragraph

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

$$
\begin{array}{l}
f_1 ::= h_{1\,1} \mid \ldots \mid h_{1\,m_1} \mid g_{1\,1} \langle\!\langle e_{1\,1} \rangle\!\rangle \mid \ldots \mid g_{1\,n_1} \langle\!\langle e_{1\,n_1} \rangle\!\rangle \\
\mathbin{\&} \ldots \mathbin{\&} \\
f_r ::= h_{r\,1} \mid \ldots \mid h_{r\,m_r} \mid g_{r\,1} \langle\!\langle e_{r\,1} \rangle\!\rangle \mid \ldots \mid g_{r\,n_r} \langle\!\langle e_{r\,n_r} \rangle\!\rangle \\
\\
\qquad \Longrightarrow \\
\\
[f_1, \ldots, f_r] \\
\mathtt{END}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{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 \\
\mathtt{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 1 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 2 The right-hand side of this transformation could have been expressed using the following notation from the mathematical toolkit, but for the desire to define the core language separately from the mathematical toolkit.

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

15.2.4 Predicate

15.2.4.1 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*.

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

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.

15.2.5 Expression

15.2.5.1 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.

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

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.

15.2.5.2 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.

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

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

15.2.5.3 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.

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

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.

15.2.5.4 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.

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

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.

15.2.5.5 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.

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

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

15.2.5.6 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.

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

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

15.2.5.7 Schema precondition expression

The value of the schema precondition expression pre e is that schema which is like schema e but without its primed and shrieked components.

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

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

15.2.5.8 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.

$$\begin{aligned}(e_1 \mathrel{:} \mathbb{P}[\sigma_1]) \fatsemi (e_2 \mathrel{:} \mathbb{P}[\sigma_2]) \mathrel{:} \mathbb{P}[\sigma] \quad \Longrightarrow \quad & \mathrm{let}\, e_3 == carrier\ [\sigma_1 \setminus \sigma];\\ & e_4 == carrier\ [\sigma_2 \setminus \sigma]\\ & \bullet\ \mathrm{let}\, e^{\bowtie} == e_4 \mathit{uniquely\ renamed}\\ & \quad \bullet\ \exists\ e^{\bowtie} \bullet (\exists\ e_3 \bullet [e_1;\ e^{\bowtie} \mid \theta\, e_3 = \theta\, e^{\bowtie}])\\ & \qquad \wedge\ (\exists\ e_4 \bullet [e_2;\ e^{\bowtie} \mid \theta\, e_4 = \theta\, e^{\bowtie}])\end{aligned}$$

It is semantically equivalent to the existential quantification of the matched pairs of primed components of e_1 and unprimed components of e_2 (as given by the signatures determined by typechecking), with those matched pairs being equated.

15.2.5.9 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.

$$\begin{aligned}(e_1 \mathrel{:} \mathbb{P}[\sigma_1]) \gg (e_2 \mathrel{:} \mathbb{P}[\sigma_2]) \mathrel{:} \mathbb{P}[\sigma] \quad \Longrightarrow \quad & \mathrm{let}\, e_3 == carrier\ [\sigma_1 \setminus \sigma];\\ & e_4 == carrier\ [\sigma_2 \setminus \sigma]\\ & \bullet\ \mathrm{let}\, e^{\bowtie} == e_4 \mathit{uniquely\ renamed}\\ & \quad \bullet\ \exists\ e^{\bowtie} \bullet (\exists\ e_3 \bullet [e_1;\ e^{\bowtie} \mid \theta\, e_3 = \theta\, e^{\bowtie}])\\ & \qquad \wedge\ (\exists\ e_4 \bullet [e_2;\ e^{\bowtie} \mid \theta\, e_4 = \theta\, e^{\bowtie}])\end{aligned}$$

It is semantically equivalent to the existential quantification of the matched pairs of shrieked components of e_1 and queried components of e_2 (as given by the signatures determined by typechecking), with those matched pairs being equated.

15.2.5.10 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 $^{+}$.

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

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

16 Semantic relations

16.1 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 1 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.

16.2 Formal definition of semantic relations

16.2.1 Specification

16.2.1.1 Sectioned specification

$$[\![\ s_1\ \ldots\ s_n\ ]\!]^{\mathcal{Z}} \quad = \quad ([\![\ \text{section}\ \mathit{prelude}\ldots\ ]\!]^{\mathcal{S}} \fatsemi [\![\ s_1\ ]\!]^{\mathcal{S}} \fatsemi \ldots \fatsemi [\![\ s_n\ ]\!]^{\mathcal{S}})\ \varnothing$$

The meaning of the Z specification $s_1\ \ldots\ s_n$ is the set of named theories to which the empty set of named theories is related by the composition of the relations between sets of named theories that denote the meaning of each section, starting with the prelude.

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

NOTE 1 The meaning of a specification is not the meaning of its last section, so as to permit several meaningful units within a single document.

16.2.2 Section

16.2.2.1 Inheriting section

NOTE 1 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.

$$[\![\ \text{section}\ \mathit{prelude}\ \text{parents}\ \texttt{END}\ d_1\ \ldots\ d_n\ ]\!]^{\mathcal{S}}$$
$$=$$
$$\lambda\ T : \mathit{Theory} \bullet \{\mathit{prelude} \mapsto ([\![\ d_1\ ]\!]^{\mathcal{D}} \fatsemi \ldots \fatsemi [\![\ d_n\ ]\!]^{\mathcal{D}})\ (\!|\ \{\varnothing\}\ |\!)\}$$

The meaning of the prelude section is given by that constant function which, whatever set of named theories it is given, returns the singleton set containing the theory named prelude, whose models are those to which the empty set of models is related by the composition of the relations between models that denote the meanings of each of its paragraphs – see clause 11 for details of those paragraphs.

$$[\![\text{ section } i \text{ parents } i_1, ..., i_m \texttt{ END } d_1 \ ... \ d_n \ ]\!]^{\mathcal{S}}$$
$$=$$
$$\lambda\ T : \mathit{Theory} \bullet T \cup \{i \mapsto$$
$$([\![\ d_1 \ ]\!]^{\mathcal{D}} \fatsemi ... \fatsemi [\![\ d_n \ ]\!]^{\mathcal{D}}) \ (\!| \ \{M_0 : T \ \mathit{prelude};\ M_1 : T \ i_1;\ ...;\ M_m : T \ i_m \bullet M_0 \cup M_1 \cup ... \cup M_m\} \ |\!)\}$$

The meaning of a section other than the prelude is the function that augments a given set of named theories with the named theory of the given section. The models in that named theory are those 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.

16.2.3 Paragraph

16.2.3.1 Given types paragraph

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

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

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 (16.2.6.1) can avoid being captured.

16.2.3.2 Axiomatic description paragraph

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

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

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.

16.2.3.3 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.

$$\begin{aligned}&[\![\ \texttt{GENAX } [i_1, ..., i_n] \ (e \ \fatsemi \ \mathbb{P}[j_1 : \tau_1;\ ...;\ j_m : \tau_m]) \texttt{ END }]\!]^{\mathcal{D}} = \\ &\quad \{M : \mathit{Model};\ u : \mathbb{W} \to \mathbb{W} \\ &\qquad \mid \forall\ w_1, ..., w_n : \mathbb{W} \bullet \exists\ w : \mathbb{W} \bullet \\ &\qquad\quad u\ (w_1, ..., w_n) \in w \\ &\qquad\quad \land\ (M \oplus \{i_1 \mapsto w_1, ..., i_n \mapsto w_n\} \cup \{\mathit{decor} \spadesuit i_1 \mapsto w_1, ..., \mathit{decor} \spadesuit i_n \mapsto w_n\}) \mapsto w \in [\![\ e \ ]\!]^{\mathcal{E}} \\ &\qquad \bullet M \mapsto M \cup \lambda\ y : \{j_1, ..., j_m\} \bullet \lambda\ x : \mathit{dom}\ u \bullet u\ x\ y\}\end{aligned}$$

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 (16.2.6.2) can avoid being captured.

16.2.3.4 Conjecture paragraph

The conjecture paragraph $\models ?\ p$ `END` expresses a property that may logically follow from the specification. It may be a starting point for a proof.

$$[\![\ \models ?\ p \texttt{ END }]\!]^{\mathcal{D}} \ = \ \mathit{id}\ \mathit{Model}$$

It relates a model to itself: the truth of p in a model does not affect the meaning of the specification.

16.2.3.5 Generic conjecture paragraph

The generic conjecture paragraph $[i_1, ..., i_n] \models? \; p$ `END` expresses a generic property that may logically follow from the specification. It may be a starting point for a proof.

$$[\![\; [i_1, ..., i_n] \models? \; p \; \texttt{END} \;]\!]^{\mathcal{D}} \quad = \quad id \; Model$$

It relates a model to itself: the truth of p in a model does not affect the meaning of the specification.

16.2.4 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.

16.2.4.1 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.

16.2.4.2 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.

16.2.4.3 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*.

16.2.4.4 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.

16.2.4.5 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.

16.2.5 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.

16.2.5.1 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.

16.2.5.2 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.

16.2.5.3 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.

16.2.5.4 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.

16.2.5.5 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.

16.2.5.6 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.

16.2.5.7 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.

16.2.5.8 Definite description expression

The value of the definite description expression $\mu\ e_1 \bullet e_2$ is the unique value of e_2 that arises whichever binding is chosen from the set that is the value of schema e_1.

$$\begin{array}{l} \{M : Model;\ t_1 : \mathbb{W} \\ \quad \mid t_1 \in [\![\ e_1\ ]\!]^{\mathcal{E}} M \\ \quad \land\ (\forall\ t_3 : [\![\ e_1\ ]\!]^{\mathcal{E}} M \bullet [\![\ e_2\ ]\!]^{\mathcal{E}}\ (M \oplus t_3) = [\![\ e_2\ ]\!]^{\mathcal{E}}\ (M \oplus t_1)) \\ \bullet\ M \mapsto [\![\ e_2\ ]\!]^{\mathcal{E}}\ (M \oplus t_1)\} \end{array} \subseteq\ [\![\ \mu\ e_1 \bullet e_2\ ]\!]^{\mathcal{E}}$$

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.

16.2.5.9 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.

16.2.5.10 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.

16.2.5.11 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 which 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.

16.2.5.12 Schema conjunction expression

The value of the schema conjunction expression $e_1 \land e_2$ is the schema resulting from merging the signatures of schemas e_1 and e_2 and conjoining their constraints.

$$[\![\ e_1 \land e_2\ \mathbin{:}\ \mathbb{P}\,\tau\ ]\!]^{\mathcal{E}} \ =\ \lambda\ M : Model \bullet \{t : [\![\ \tau\ ]\!]^{\mathcal{T}} M;\ t_1 : [\![\ e_1\ ]\!]^{\mathcal{E}} M;\ t_2 : [\![\ e_2\ ]\!]^{\mathcal{E}} M \mid t_1 \cup t_2 = t \bullet t\}$$

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.

16.2.5.13 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.

16.2.5.14 Schema renaming expression

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.

$$\begin{aligned}[\![\ e\ [j_1\ /\ i_1, ..., j_n\ /\ i_n]\ ]\!]^{\mathcal{E}} \ = \ & \lambda\ M : Model \bullet \\ & \{t_1 : [\![\ e\ ]\!]^{\mathcal{E}} M;\ t_2 : \mathbb{W} \mid \\ & \quad t_2 = (id\ (dom\ t_1) \oplus \{i_1 \mapsto j_1, ..., i_n \mapsto j_n\})^{\sim} \fatsemi t_1 \\ & \quad \land\ t_2 \in (\_ \pfun \_) \\ & \bullet\ t_1\}\end{aligned}$$

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.

16.2.6 Type

The value of a type is its carrier set.

NOTE 1 For an expression e with a defined value, $[\![\ e\ \fatsemi\ \tau\ ]\!]^{\mathcal{E}} \in [\![\ \tau\ ]\!]^{\mathcal{T}}$.

16.2.6.1 Given type

$$[\![\ \texttt{GIVEN}\ i\ ]\!]^{\mathcal{T}} \ = \ \lambda\ M : Model \bullet M\ (decor\ \heartsuit\ i)$$

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.

16.2.6.2 Generic parameter type

$$[\![\ \texttt{GENTYPE}\ i\ ]\!]^{\mathcal{T}} \ = \ \lambda\ M : Model \bullet M\ (decor\ \spadesuit\ i)$$

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.

16.2.6.3 Set type

$$[\![\ \mathbb{P}\tau\ ]\!]^{\mathcal{T}} \ = \ \lambda\ M : Model \bullet \mathbb{P}\ ([\![\ \tau\ ]\!]^{\mathcal{T}} M)$$

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.

16.2.6.4 Cartesian product type

$$[\![\ \tau_1 \times ... \times \tau_n\ ]\!]^{\mathcal{T}} \ = \ \lambda\ M : Model \bullet ([\![\ \tau_1\ ]\!]^{\mathcal{T}} M) \times ... \times ([\![\ \tau_n\ ]\!]^{\mathcal{T}} M)$$

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.

16.2.6.5 Schema type

$$[\![\ [i_1 : \tau_1;\ ...;\ i_n : \tau_n]\ ]\!]^{\mathcal{T}} \ = \ \lambda\ M : Model \bullet \{t : \{i_1,\ ...,\ i_n\} \to \mathbb{W} \mid t\ i_1 \in [\![\ \tau_1\ ]\!]^{\mathcal{T}} M \land ... \land t\ i_n \in [\![\ \tau_n\ ]\!]^{\mathcal{T}} M \bullet t\}$$

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.

16.2.6.6 Generic type

$$[\![\ [i_1, ..., i_n]\tau\ ]\!]^{\mathcal{T}} \ = \ \lambda\ M : Model \bullet \{g : \mathbb{W} \to \mathbb{W} \mid \forall\ w_1, ..., w_n : \mathbb{W} \bullet g(w_1,\ ...,\ w_n) \in [\![\ \tau\ ]\!]^{\mathcal{T}}\ (M \oplus \{i_1 \mapsto w_1,\ ...,\ i_n \mapsto w_n\}) \bullet g\}$$

The semantic value of the generic type $[i_1, ..., i_n]\tau$, given a model M, is the set of functions from the tuple of semantic values of instantiating types to the semantic value of a type.

NOTE 1 Variable types do not appear in the type annotations of well-typed specifications, so do not need to be given semantics here.

NOTE 2 Generic types appear at only the outermost level of a type, so the variables w_k in the last relation need range over only $\mathbb{W}$ not $\mathbb{U}$.

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.

Annex A
(normative)

Mark-ups

A.1 Introduction

Not all systems support 16-bit Unicode [7], the definitive representation of Z characters (clause 6). A mark-up is a mapping to (or from) the Unicode representation. This annex defines two mark-ups based on 7-bit ASCII [5]:

- a LaTeX [11] mark-up, suitable for processing by that tool to render Z characters in their mathematical form;
- an email, or lightweight ASCII, mark-up, suitable for rendering Z characters on a low resolution device, such as an ASCII-character-based terminal, or in email conversation.

The mark-ups described in this annex show how to translate between a 'mark-up token' (string of ASCII mark-up characters) into the corresponding string of Z characters. Remaining individual mark-up characters that do not form a special mark-up token (such as digits, Latin letters, and much punctuation) are converted directly to the corresponding Z character, from ASCII-xy to Unicode U+00xy.

A chosen mark-up language may also be used to specify a particular rendering for the characters, for example, bold or italic.

A.2 LaTeX mark-up

A LaTeX command is a backslash '\' followed by a string of alphabetic characters (up to the first non-alphabetic character), or by a single non-alphabetic character.

A.2.1 Letter characters

A.2.1.1 Greek alphabet characters

Only the minimal subset of Greek alphabet defined in 6.2 need be supported by an implementation. LaTeX does not support upper case Greek letters that look like Roman counterparts. Those Greek characters that shall be supported shall use the mark-up given here.

LaTeX command	Z character string
`\Delta`	Δ
`\Xi`	Ξ
`\theta`	θ
`\lambda`	λ
`\mu`	μ

A.2.1.2 Other Z core language letter characters

LaTeX command	Z character string
`\arithmos`	$\mathbb{A}$
`\nat`	$\mathbb{N}$
`\power`	$\mathbb{P}$

A.2.2 Special characters

A.2.2.1 Special characters except Box characters

LaTeX command	Z character string
`\_`	_
`\{`	{
`\}`	}
`\ldata`	⦇
`\rdata`	⦈
`\lblot`	⦉
`\rblot`	⦊

Subscripts and superscripts shall be marked up as follows:

LaTeX command	Z character string
`^` ⟨single LaTeX token⟩	↗⟨ Z string ⟩ ↙
`^{` ⟨LaTeX tokens⟩ `}`	↗⟨ Z string ⟩ ↙
`_` ⟨single LaTeX token⟩	↘⟨ Z string ⟩ ↖
`_{` ⟨LaTeX tokens⟩ `}`	↘⟨ Z string ⟩ ↖

EXAMPLE 1 LaTeX mark-up `x^1` corresponds to Z character string '$x \nearrow 1 \swarrow$', which may be rendered 'x^1'
LaTeX mark-up `x^{1}` corresponds to Z character string '$x \nearrow 1 \swarrow$', which may be rendered 'x^1'
LaTeX mark-up `x^{\Delta S}` corresponds to Z character string '$x \nearrow \Delta S \swarrow$', which may be rendered '$x^{\Delta S}$'
LaTeX mark-up `\exists_1` corresponds to Z character string '$\exists \searrow 1 \nwarrow$', which may be rendered '$\exists_1$'
LaTeX mark-up `\exists_{1}` corresponds to Z character string '$\exists \searrow 1 \nwarrow$', which may be rendered '$\exists_1$'
LaTeX mark-up `\exists_{\Delta S}` corresponds to Z character string '$\exists \searrow \Delta S \nwarrow$', which may be rendered '$\exists_{\Delta S}$'
LaTeX mark-up `x_a^b` corresponds to Z character string '$x \searrow a \nwarrow \nearrow b \swarrow$', which may be rendered '${x_a}^b$'
LaTeX mark-up `x_{a^b}` corresponds to Z character string '$x \searrow a \nearrow b \swarrow \nwarrow$', which may be rendered 'x_{a^b}'

A.2.2.2 Box characters

The `ENDCHAR` character is used to mark the end of a `Paragraph`. The `NLCHAR` character is used to mark a hard newline (see 7.5). Different implementations may represent these characters in different ways.

The box characters are described in A.2.6, on paragraph mark-up.

A.2.3 Symbol characters (except mathematical toolkit characters)

LaTeX command	Z character string
`\models`	⊨
`\land`	∧
`\lor`	∨
`\implies`	⇒
`\iff`	⇔
`\lnot`	¬
`\forall`	∀
`\exists`	∃
`\cross`	×
`\in`	∈
`@`	•

\hide	$\backslash$
\project	$\upharpoonright$
\semi	$\fatsemi$
\pipe	$\gg$

A.2.4 Core tokens

The Roman typeface used for core tokens in this International Standard is obtained using the mark-up defined in A.2.9, with the following exceptions where there would otherwise be clashes with LaTeX keywords.

LaTeX command	Z character string
\IF	if
\THEN	then
\ELSE	else
\LET	let
\zsection	section

A.2.5 Mathematical toolkit characters and tokens

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

LaTeX command	Z character string
\rel	$\leftrightarrow$
\fun	$\rightarrow$
\neq	$\neq$
\notin	$\notin$
\emptyset	$\varnothing$
\subseteq	$\subseteq$
\subset	$\subset$
\cup	$\cup$
\cap	$\cap$
\setminus	$\setminus$
\symdiff	$\ominus$
\bigcup	$\bigcup$
\bigcap	$\bigcap$
\finset	$\mathbb{F}$
\mapsto	$\mapsto$
\comp	$\fatsemi$
\circ	$\circ$
\dres	$\lhd$
\rres	$\rhd$
\ndres	$\unlhd$
\nrres	$\unrhd$
\inv	$^{\sim}$
\limg	$(\!\mid$
\rimg	$\mid\!)$
\oplus	$\oplus$

\plus	$\nearrow + \swarrow$
\star	$\nearrow * \swarrow$
\pfun	$\pfun$
\pinj	$\pinj$
\inj	$\inj$
\psurj	$\psurj$
\surj	$\surj$
\bij	$\bij$
\ffun	$\ffun$
\finj	$\finj$
\num	$\num$
\negate	-
-	$-$
\leq	$\leq$
<	$<$
\geq	$\geq$
>	$>$
\upto	$\upto$
\#	$\#$
\langle	$\langle$
\rangle	$\rangle$
\cat	$\cat$
\extract	$\extract$
\filter	$\filter$
\dcat	$\dcat$

A.2.6 Paragraph mark-up

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.

For boxed paragraphs, the `\begin{xxx}` command indicates some box character, while for other paragraphs the `\begin{xxx}` command is Z whitespace. Any middle line in a boxed paragraph is marked-up using the `\where` LaTeX command, which corresponds to the Z | character. The `\end{xxx}` command represents the Z `ENDCHAR` character.

A.2.6.1 Axiomatic description paragraph mark-up

```
\begin{axdef}
DeclPart
\where
Predicate
\end{axdef}
```

A.2.6.2 Schema definition paragraph mark-up

```
\begin{schema}{NAME}
DeclPart
\where
Predicate
\end{schema}
```

A.2.6.3 Generic axiomatic description paragraph mark-up

```
\begin{gendef}[Formals]
```

```
DeclPart
\where
Preidcate
\end{gendef}
```

A.2.6.4 Generic schema definition paragraph mark-up

```
\begin{schema}{NAME}[Formals]
DeclPart
\where
Predicate
\end{schema}
```

A.2.6.5 Free type paragraph mark-up

```
\begin{syntax}
Freetype, { & , Freetype }
\end{syntax}
```

A.2.6.6 Other paragraph mark-up

All other paragraphs are enclosed in a pair of `\begin{zed}` and `\end{zed}` commands, which are converted to Z white space.

A.2.7 LaTeX whitespace mark-up

LaTeX has 'hard' white space (explicit LaTeX mark-up) and 'soft' white space (ASCII white space characters such as space, tab, and new line).

The hard white space is converted as follows:

LaTeX command	**Z character string**
`{`	(empty)
`}`	(empty)
(tab)	`SPACE`
`~`	`SPACE`
`\,`	`SPACE`
`\!`	`SPACE`
`\`(space)	`SPACE`
`\;`	`SPACE`
`\:`	`SPACE`
`\t1`	`SPACE`
`\t2`	`SPACE`
`\t3`	`SPACE`
`\t4`	`SPACE`
`\t5`	`SPACE`
`\t6`	`SPACE`
`\t7`	`SPACE`
`\t8`	`SPACE`
`\t9`	`SPACE`
`\\`	`NLCHAR`
`\also`	`NLCHAR`

The conversion of LaTeX Greek characters shall consume any immediately following soft white space. The conversion of LaTeX symbol characters shall preserve any following soft white space. Any remaining soft white space shall be converted to the `SPACE` character.

EXAMPLE 1 The LaTeX command '`\Delta S`' converts to the Z string 'ΔS'.

EXAMPLE 2 The LaTeX command '`\Delta~S`' converts to the Z string '$\Delta\ S$'.

EXAMPLE 3 The LaTeX command '`\power S`' converts to the Z string '$\mathbb{P}\ S$'.

A.2.8 Introducing new Z characters

New Z characters are introduced by

```
%%Zchar \LaTeXcommand U+nnnn
%%Zstring \LaTeXcommand {Zstring} [in LaTeX mark-up]
```

EXAMPLE 1
```
%%Zchar \sqsubseteq U+2291
%%Zstring \nattwo {\nat_2}
```

A.2.9 Remaining LaTeX mark-up

Any remaining LaTeX command names enclosed in braces, '`{\aToken}`', shall be converted to the equivalent Z character string with the braces and leading backslash removed, as '`aToken`'.

Any remaining LaTeX command names shall be converted to the equivalent Z character string with the leading backslash removed, and with a `SPACE` character added at the beginning and end, as '` aToken `'.

EXAMPLE 1 LaTeX mark-up: '`{\dom}s`', Z character string: '`doms`'.
LaTeX mark-up: '`\dom s`', Z character string: '` dom s`'.

EXAMPLE 2 LaTeX mark-up: `\IF \disjoint a \THEN x = y \mod z \ELSE x = y \div z`
Z character string: `if disjoint a then x = y mod z else x = y div z`
a possible rendering: if $disjoint\ a$ then $x = y\ mod\ z$ else $x = y\ div\ z$

A.3 Email mark-up

This email mark-up is designed primarily as a human-readable lightweight mark-up, rather than for processing by tools. The character '`%`' is used to flag a special string, for example '$\times$' as '`%x`', and disambiguate it from, for example, the name '`x`'. This flag character may be omitted to reduce clutter, if there is no danger of ambiguity (for the human reader).

A.3.1 Letter characters

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 shall be supported shall use the mark-up given here.

Email string	**Z character string**
`%Delta%`	Δ
`%Xi%`	Ξ
`%theta%`	θ
`%lambda%`	λ
`%mu%`	μ

A.3.1.2 Other Z core language letter characters

Email string	Z character string
%arithmos	$\mathbb{A}$
%N	$\mathbb{N}$
%P	$\mathbb{P}$

A.3.2 Special characters

A.3.2.1 Special characters except Box characters

Email string	Z character string
,	′
!	!
?	?
/^	$\nearrow$
v/	$\swarrow$
\v	$\searrow$
^\	$\nwarrow$
-	-
(	(
)	)
[	[
]	]
{	{
}	}
<<	$\langle\!\langle$
>>	$\rangle\!\rangle$
<|	$\langle\!\vert$
|>	$\vert\!\rangle$

A.3.2.2 Box characters

The ENDCHAR character is used to mark the end of a Paragraph. The NLCHAR character is used to mark a hard newline (see 7.5). Different implementations may represent these characters in different ways.

The email form of the box characters mimicks the mathematical form, as various boxes drawn around the text.

EXAMPLE 1 An example of the mathematical rendering of an axiomatic paragraph, illustrating the AX token, comprising AXCHAR:

```
+..
  succ : %N --> %N
|--
  %A n : %N @ succ n = n + 1
-..
```

EXAMPLE 2 An example of the mathematical rendering of an schema paragraph, illustrating the SCH token, comprising SCHCHAR:

```
+-- AddName ---
  %Delta System
  n? : NAME
```

```
|--
  name ' = name %u n?
---
```

EXAMPLE 3 An example of the mathematical rendering of a generic axiomatic paragraph, illustrating the `GENAX` token, comprising `AXCHAR`, `GENCHAR`:

```
+== [X] ===
  _ %u _ : %P X %x %P X --> %P X
|--
  %A a,b: %P X @ a %u b = { x:X | x %e a \/ x %e b }
-==
```

EXAMPLE 4 An example of the mathematical rendering of a generic schema paragraph, illustrating the `GENSCH` token, comprising `SCHCHAR`, `GENCHAR`:

```
+-- BoundedStack[X] ---
  stack : seq X
  maxSize : %N
|--
  # stack <= maxSize
---
```

A.3.3 Symbol characters (except mathematical toolkit characters)

Email string	Z character string
|	$\mid$
|=	$\models$
/\	$\land$
\/	$\lor$
==>	$\Rightarrow$
<=>	$\Leftrightarrow$
%not	$\neg$
%A	$\forall$
%E	$\exists$
%x	$\times$
/	$/$
=	$=$
%e	$\in$
:	:
;	;
,	,
.	.
@	$\bullet$
%\	$\setminus$
%|	$\upharpoonright$
%%;	$\overset{\circ}{9}$
%%>>	$\gg$

A.3.4 Mathematical toolkit characters and tokens

The mathematical toolkit 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 are not listed here.

Email string	Z character string
<-->	$\leftrightarrow$
-->	$\rightarrow$
/=	$\neq$
%/e	$\notin$
(/)	$\varnothing$
%c_	$\subseteq$
%c	$\subset$
%u	$\cup$
%n	$\cap$
\	$\setminus$
(-)	$\ominus$
%uu	$\bigcup$
%nn	$\bigcap$
%F	$\mathbb{F}$
|-->	$\mapsto$
%;	$\fatsemi$
%o	$\circ$
<:	$\lhd$
:>	$\rhd$
<-:	$\unlhd$
:->	$\unrhd$
~	$\sim$
(|	$(\!\mid$
|)	$\mid\!)$
(+)	$\oplus$
%+	$+$
%*	$*$
-|->	$\pfun$
>-|->	$\fun$
>-->	$\inj$
-|->>	$\psurj$
-->>	$\surj$
>-->>	$\bij$
-||->	$\finj$
>-||->	$\ffun$
%Z	$\mathbb{Z}$
%-	-
-	$-$
<=	$\leq$
<	$<$
>=	$\geq$
>	$>$
#	$\#$

%<	⟨
%>	⟩
^	⌢
/|	⌈
|\	⌉

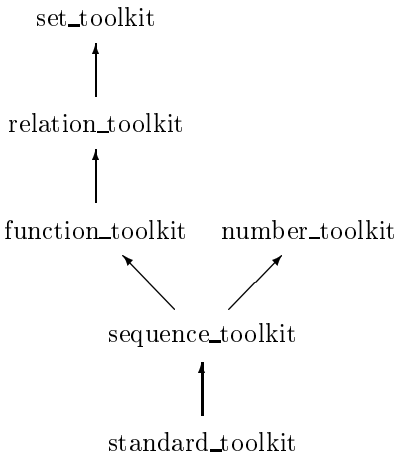
Annex B
(normative)

Mathematical toolkit

B.1 Introduction

The mathematical toolkit is an optional extension to the compulsory 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 specific ones to be reused or replaced. For example, *sequence_toolkit* could be bypassed if not needed, and *function_toolkit* could be replaced without affecting *number_toolkit*.

A specification without a section header has section *standard_toolkit* as parent by default.

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.

NOTE 1 The $\mathbb{P}$ symbol is established as a generic operator by the prelude.

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 80 ($\mathbb{F}\_$)

$$\mathbb{F}\,X == \bigcap\{\, A : \mathbb{P}\,\mathbb{P}\,X \mid \varnothing \in A \land (\forall\, a : A;\ x : X \bullet a \cup \{x\} \in A) \,\}$$

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

$$[X, Y]$$
$$first : X \times Y \to X$$
$$\forall\, x : X;\ y : Y \bullet first\ (x, y) = x$$

For any ordered pair (x, y), $first\ (x, y)$ is the x component of the pair.

B.5.2 Second component projection

$$[X, Y]$$
$$second : X \times Y \to Y$$
$$\forall\, x : X;\ y : Y \bullet second\ (x, y) = y$$

For any ordered pair (x, y), $second\ (x, y)$ is the y 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 80 $(id \_)$

$$id\, X == \{ x : X \bullet x \mapsto x \}$$

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 61 rightassoc (_ ◁ _)

$$[X, Y]$$
$$\_ \lhd \_ : \mathbb{P} X \times (X \leftrightarrow Y) \to (X \leftrightarrow Y)$$
$$\forall a : \mathbb{P} X;\ r : X \leftrightarrow Y \bullet a \lhd r = \{ p : r \mid p.1 \in a \}$$

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 (_ ▷ _)

$$[X, Y]$$
$$\_ \rhd \_ : (X \leftrightarrow Y) \times \mathbb{P} Y \to (X \leftrightarrow Y)$$
$$\forall r : X \leftrightarrow Y;\ b : \mathbb{P} Y \bullet r \rhd b = \{ p : r \mid p.2 \in b \}$$

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 61 rightassoc (_ ⩤ _)

$$[X, Y]$$
$$\_ \ndres \_ : \mathbb{P} X \times (X \leftrightarrow Y) \to (X \leftrightarrow Y)$$
$$\forall a : \mathbb{P} X;\ r : X \leftrightarrow Y \bullet a \ndres r = \{ p : r \mid p.1 \notin a \}$$

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 (_ ⩥ _)

$$[X, Y]$$
$$\_ \rndres \_ : (X \leftrightarrow Y) \times \mathbb{P} Y \to (X \leftrightarrow Y)$$
$$\forall r : X \leftrightarrow Y;\ b : \mathbb{P} Y \bullet r \rndres b = \{ p : r \mid p.2 \notin b \}$$

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 90 (_ ~)

$$[X, Y]$$
$$\_^{\sim} : (X \leftrightarrow Y) \to (Y \leftrightarrow X)$$
$$\forall r : X \leftrightarrow Y \bullet r^{\sim} = \{ p : r \bullet p.2 \mapsto p.1 \}$$

The inverse of a relation is the relation obtained by reversing every ordered pair in the relation.

B.5.14 Relational image

function 90 (_ (| _ |))

$$[X, Y]$$
$$\_ (| \_ |) : (X \leftrightarrow Y) \times \mathbb{P} X \to \mathbb{P} Y$$
$$\forall r : X \leftrightarrow Y;\ a : \mathbb{P} X \bullet r(| a |) = \{ p : r \mid p.1 \in a \bullet p.2 \}$$

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 90 (_ $^+$)

$$[X]$$
$$\_ ^+ : (X \leftrightarrow X) \to (X \leftrightarrow X)$$
$$\forall r : X \leftrightarrow X \bullet r^+ = \bigcap \{ s : X \leftrightarrow X \mid r \subseteq s \land r \fatsemi s \subseteq s \}$$

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 90 (_ *)

$$[X]$$
$$\_ ^* : (X \leftrightarrow X) \to (X \leftrightarrow X)$$
$$\forall r : X \leftrightarrow X \bullet r^* = r^+ \cup id\ X$$

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 $(\_ \pfun \_)$

$$X \pfun Y == \{ f : X \rel Y \mid \forall p, q : f \mid p.1 = q.1 \bullet p.2 = q.2 \}$$

$X \pfun Y$ 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 $(\_ \pinj \_)$

$$X \pinj Y == \{ f : X \rel Y \mid \forall p, q : f \bullet p.1 = q.1 \Leftrightarrow p.2 = q.2 \}$$

$X \pinj Y$ 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 $(\_ \inj \_)$

$$X \inj Y == (X \pinj Y) \cap (X \fun Y)$$

$X \inj Y$ 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 $(\_ \surj \_)$

$$X \surj Y == (X \psurj Y) \cap (X \fun Y)$$

$X \surj Y$ 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 $(\_ \bij \_)$

$$X \bij Y == (X \surj Y) \cap (X \inj Y)$$

$X \bij Y$ 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 (_ $\pfun$ _)

$$X \pfun Y == (X \fun Y) \cap \mathbb{F}(X \times Y)$$

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 (_ $\pinj$ _)

$$X \pinj Y == (X \pfun Y) \cap (X \inj Y)$$

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* _)

$$[L, X]$$

$$\mathit{disjoint}\ \_ : \mathbb{P}(L \rel \mathbb{P}\, X)$$

$$\forall f : L \rel \mathbb{P}\, X \bullet \mathit{disjoint}\, f \Leftrightarrow (\forall p, q : f \mid p \neq q \bullet p.2 \cap q.2 = \varnothing)$$

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* _)

$$[L, X]$$

$$\_\ \mathit{partition}\ \_ : (L \rel \mathbb{P}\, X) \rel \mathbb{P}\, X$$

$$\forall f : L \rel \mathbb{P}\, X;\ a : \mathbb{P}\, X \bullet f\ \mathit{partition}\ a \Leftrightarrow \mathit{disjoint}\, f \land \bigcup(\mathit{ran}\, f) = a$$

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

$$\mathit{succ} : \mathbb{N} \fun \mathbb{N}$$

$$\forall n : \mathbb{N} \bullet \mathit{succ}\ n = n + 1$$

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 50 (- _)

$$
\begin{array}{|l}
\hline
\text{- \_} : \mathbb{P}(\mathbb{A} \times \mathbb{A}) \\
\hline
\forall x, y : \mathbb{Z} \bullet \exists_1 z : \mathbb{Z} \bullet ((x, y), z) \in (\_ + \_) \\
\forall x : \mathbb{Z} \bullet \exists_1 y : \mathbb{Z} \bullet (x, y) \in (\text{- } \_) \\
\forall i, j, k : \mathbb{Z} \bullet \\
\quad (i + j) + k = i + (j + k) \\
\quad \land\ i + j = j + i \\
\quad \land\ i + \text{-}i = 0 \\
\quad \land\ i + 0 = i \\
\mathbb{Z} = \{z : \mathbb{A} \mid \exists x : \mathbb{N} \bullet z = x \lor z = \text{-}x\} \\
\hline
\end{array}
$$

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 (- _) as the inverse operation and 0 as the identity element.

NOTE 1 If *function_toolkit* notation were exploited, the negation operator could be defined as follows.

$$
\begin{array}{|l}
\hline
\text{- \_} : \mathbb{A} \pfun \mathbb{A} \\
\hline
(\mathbb{Z} \times \mathbb{Z}) \lhd (\_ + \_) \in \mathbb{Z} \times \mathbb{Z} \fun \mathbb{Z} \\
\mathbb{Z} \lhd (\text{- } \_) \in \mathbb{Z} \fun \mathbb{Z} \\
\forall i, j, k : \mathbb{Z} \bullet \\
\quad (i + j) + k = i + (j + k) \\
\quad \land\ i + j = j + i \\
\quad \land\ i + \text{-}i = 0 \\
\quad \land\ i + 0 = i \\
\forall h : \mathbb{P}\,\mathbb{Z} \bullet \\
\quad 1 \in h \land (\forall i, j : h \bullet i + j \in h \land \text{-}i \in h) \\
\qquad \Rightarrow h = \mathbb{Z} \\
\hline
\end{array}
$$

B.7.4 Subtraction

function 30 leftassoc (_ − _)

$$
\begin{array}{|l}
\hline
\_ - \_ : \mathbb{P}((\mathbb{A} \times \mathbb{A}) \times \mathbb{A}) \\
\hline
\forall x, y : \mathbb{Z} \bullet \exists_1 z : \mathbb{Z} \bullet ((x, y), z) \in (\_ - \_) \\
\forall i, j : \mathbb{Z} \bullet i - j = i + \text{-}j \\
\hline
\end{array}
$$

Subtraction is a function whose domain includes all pairs of integers. For all integers i and j, $i - j$ is equal to $i + \text{-}j$.

NOTE 1 If *function_toolkit* notation were exploited, the subtraction operator could be defined as follows.

$$
\begin{array}{|l}
\hline
\_ - \_ : \mathbb{A} \times \mathbb{A} \pfun \mathbb{A} \\
\hline
(\mathbb{Z} \times \mathbb{Z}) \lhd (\_ - \_) \in \mathbb{Z} \times \mathbb{Z} \fun \mathbb{Z} \\
\forall i, j : \mathbb{Z} \bullet i - j = i + \text{-}j \\
\hline
\end{array}
$$

B.7.5 Less-than-or-equal

relation $(\_ \leq \_)$

$$\_ \leq \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A})$$
$$\forall i, j : \mathbb{Z} \bullet i \leq j \Leftrightarrow j - i \in \mathbb{N}$$

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 $(\_ < \_)$

$$\_ < \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A})$$
$$\forall i, j : \mathbb{Z} \bullet i < j \Leftrightarrow i + 1 \leq j$$

For all integers i and j, $i < j$ if and only if $i + 1 \leq j$.

B.7.7 Greater-than-or-equal

relation $(\_ \geq \_)$

$$\_ \geq \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A})$$
$$\forall i, j : \mathbb{Z} \bullet i \geq j \Leftrightarrow j \leq i$$

For all integers i and j, $i \geq j$ if and only if $j \leq i$.

B.7.8 Greater-than

relation $(\_ > \_)$

$$\_ > \_ : \mathbb{P}(\mathbb{A} \times \mathbb{A})$$
$$\forall i, j : \mathbb{Z} \bullet i > j \Leftrightarrow j < i$$

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 $(\_ * \_)$

$$
\begin{array}{|l}
\_ * \_ : \mathbb{P}((\mathbb{A} \times \mathbb{A}) \times \mathbb{A}) \\
\hline
\forall x, y : \mathbb{Z} \bullet \exists_1 z : \mathbb{Z} \bullet ((x, y), z) \in (\_ * \_) \\
\forall i, j, k : \mathbb{Z} \bullet \\
\quad (i * j) * k = i * (j * k) \\
\quad \land\ i * j = j * i \\
\quad \land\ i * (j + k) = i * j + i * k \\
\quad \land\ 0 * i = 0 \\
\quad \land\ 1 * i = i
\end{array}
$$

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 1 If *function_toolkit* notation were exploited, the multiplication operator could be defined as follows.

$$
\begin{array}{|l}
\_ * \_ : (\mathbb{A} \times \mathbb{A}) \pfun \mathbb{A} \\
\hline
(\mathbb{Z} \times \mathbb{Z}) \lhd (\_ * \_) \in \mathbb{Z} \times \mathbb{Z} \fun \mathbb{Z} \\
\forall i, j, k : \mathbb{Z} \bullet \\
\quad (i * j) * k = i * (j * k) \\
\quad \land\ i * j = j * i \\
\quad \land\ i * (j + k) = i * j + i * k \\
\quad \land\ 0 * i = 0 \\
\quad \land\ 1 * i = i
\end{array}
$$

B.7.12 Division, modulus

function 40 leftassoc $(\_ \mathit{div} \_)$
function 40 leftassoc $(\_ \mathit{mod} \_)$

$$
\begin{array}{|l}
\_ \mathit{div} \_, \_ \mathit{mod} \_ : \mathbb{P}((\mathbb{A} \times \mathbb{A}) \times \mathbb{A}) \\
\hline
\forall x : \mathbb{Z};\ y : \mathbb{Z}_1 \bullet \exists_1 z : \mathbb{Z} \bullet ((x, y), z) \in (\_ \mathit{div} \_) \\
\forall x : \mathbb{Z};\ y : \mathbb{Z}_1 \bullet \exists_1 z : \mathbb{Z} \bullet ((x, y), z) \in (\_ \mathit{mod} \_) \\
\forall i : \mathbb{Z};\ j : \mathbb{Z}_1 \bullet \\
\quad i = (i \mathbin{\mathit{div}} j) * j + i \mathbin{\mathit{mod}} j \\
\quad \land\ (0 \leq i \mathbin{\mathit{mod}} j < j \lor j < i \mathbin{\mathit{mod}} j \leq 0)
\end{array}
$$

For all integers i and non-zero integers j, the pair (i, j) is in the domain of $\_\mathit{div}\_$ and of $\_\mathit{mod}\_$, and $i \mathbin{\mathit{div}} j$ and $i \mathbin{\mathit{mod}} j$ have integer values.

When not zero, $i \mathbin{\mathit{mod}} j$ has the same sign as j. This means that $i \mathbin{\mathit{div}} j$ is the largest integer no greater than the rational number i/j.

NOTE 1 If *function_toolkit* notation were exploited, the division and modulus operators could be defined as follows.

$$\_ \mathit{div} \_, \_ \mathit{mod} \_ : \mathbb{A} \times \mathbb{A} \pfun \mathbb{A}$$

$$(\mathbb{Z} \times \mathbb{Z}_1) \domres (\_ \mathit{div} \_) \in \mathbb{Z} \times \mathbb{Z}_1 \fun \mathbb{Z}$$
$$(\mathbb{Z} \times \mathbb{Z}_1) \domres (\_ \mathit{mod} \_) \in \mathbb{Z} \times \mathbb{Z}_1 \fun \mathbb{Z}$$
$$\forall i : \mathbb{Z}; j : \mathbb{Z}_1 \bullet$$
$$\quad i = (i \mathit{div} j) * j + i \mathit{mod} j$$
$$\quad \land (0 \leq i \mathit{mod} j < j \lor j < i \mathit{mod} j \leq 0)$$

B.8 Sequences

section *sequence_toolkit* parents *function_toolkit*, *number_toolkit*

B.8.1 Number range

function 20 leftassoc (_ .. _)

$$\_ \,..\, \_ : \mathbb{A} \times \mathbb{A} \pfun \mathbb{P}\, \mathbb{A}$$

$$(\mathbb{Z} \times \mathbb{Z}) \domres (\_ \,..\, \_) \in \mathbb{Z} \times \mathbb{Z} \fun \mathbb{P}\mathbb{Z}$$
$$\forall i, j : \mathbb{Z} \bullet i \,..\, j = \{ k : \mathbb{Z} \mid i \leq k \leq j \}$$

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

$$[X]$$

$$\mathit{iter} : \mathbb{N} \fun (X \rel X) \fun (X \rel X)$$

$$\forall r : X \rel X \bullet \mathit{iter}\ 0\ r = \mathit{id}\ X$$
$$\forall r : X \rel X;\ n : \mathbb{N} \bullet \mathit{iter}(n+1) r = r \fatsemi (\mathit{iter}\ n\ r)$$

iter is the iteration function for a relation. The iteration of a relation $r : X \rel X$ zero times is the identity relation on X. The iteration of a relation $r : X \rel X$ $n+1$ times is the composition of the relation with its iteration n times.

function 90 (_ ↗ _ ↙)

$$[X]$$

$$\_^{\sim} : (X \rel X) \fun \mathbb{Z} \fun (X \rel X)$$

$$\forall r : X \rel X;\ n : \mathbb{N} \bullet r^n = \mathit{iter}\ n\ r$$

iter n r may be written as r^n.

B.8.3 Number of members of a set

$$[X]$$

$$\# : \mathbb{F}\, X \fun \mathbb{N}$$

$$\forall a : \mathbb{F}\, X \bullet \# a = (\mu\, n : \mathbb{N} \mid (\exists f : 1 \,..\, n \inj a \bullet \mathit{ran} f = a))$$

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

$$min : \mathbb{P}\,\mathbb{A} \pfun \mathbb{A}$$

$$\mathbb{P}\,\mathbb{Z} \ldres min = \{\, a : \mathbb{P}\,\mathbb{Z};\ m : \mathbb{Z} \mid m \in a \land (\forall\, n : a \bullet m \leq n) \bullet a \mapsto m \,\}$$

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

$$max : \mathbb{P}\,\mathbb{A} \pfun \mathbb{A}$$

$$\mathbb{P}\,\mathbb{Z} \ldres max = \{\, a : \mathbb{P}\,\mathbb{Z};\ m : \mathbb{Z} \mid m \in a \land (\forall\, n : a \bullet n \leq m) \bullet a \mapsto m \,\}$$

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 Items

$$[L, X]$$

$$items : (L \pfun X) \to X \pfun \mathbb{N}_1$$

$$\forall f : L \pfun X \bullet items\ f = \{\, x : ran\, f \bullet x \mapsto \#(f \rhd \{x\}) \,\}$$

The items of a finite indexed set, f, is a function from the values in the range of f to the number of the occurrences of that value in the range of f.

B.8.7 Finite sequences

generic 80 ($seq\ \_$)

$$seq\, X == \{\, f : \mathbb{N} \pfun X \mid dom\, f = 1 \,..\, \#f \,\}$$

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 $1 \,..\, n$, for some n, to elements of X.

B.8.8 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.8.9 Injective sequences

generic 80 ($iseq\ \_$)

$$iseq\, X == seq\, X \cap (\mathbb{N} \pinj X)$$

$iseq\, X$ is the set of all injective finite sequences of values of X, that is, of finite sequences over X that are also injections.

B.8.10 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.8.11 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.8.12 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.8.13 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.8.14 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.8.15 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.8.16 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.8.17 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.8.18 Extraction

function 41 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.8.19 Filtering

function 40 leftassoc $(\_ \upharpoonright \_)$

$$[X]$$
$$\_ \upharpoonright \_ : seq X \times \mathbb{P} X \to seq X$$
$$\forall s : seq X;\ a : \mathbb{P} X \bullet s \upharpoonright a = squash(s \rhd a)$$

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.8.20 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.8.21 Suffix relation

relation $(\_ \mathit{suffix} \_)$

$$[X]$$
$$\_ \mathit{suffix} \_ : \mathit{seq}\, X \leftrightarrow \mathit{seq}\, X$$
$$\forall s, t : \mathit{seq}\, X \bullet s \mathit{\ suffix\ } t \Leftrightarrow (\exists u : \mathit{seq}\, X \bullet u \frown s = t)$$

A sequence s is a suffix of another sequence t if it forms the end portion of t.

B.8.22 Infix relation

relation $(\_ \mathit{infix} \_)$

$$[X]$$
$$\_ \mathit{infix} \_ : \mathit{seq}\, X \leftrightarrow \mathit{seq}\, X$$
$$\forall s, t : \mathit{seq}\, X \bullet s \mathit{\ infix\ } t \Leftrightarrow (\exists u, v : \mathit{seq}\, X \bullet u \frown s \frown v = t)$$

A sequence s is an infix of another sequence t if it forms a mid portion of t.

B.8.23 Distributed concatenation

$$[X]$$
$$\frown / : \mathit{seq}\, \mathit{seq}\, X \to \mathit{seq}\, X$$
$$\frown / \ \langle\,\rangle = \langle\,\rangle$$
$$\forall s : \mathit{seq}\, X \bullet \frown / \ \langle s \rangle = s$$
$$\forall q, r : \mathit{seq}\, \mathit{seq}\, X \bullet \frown / (q \frown r) = (\frown / \ q) \frown (\frown / \ r)$$

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
(informative)

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 following material, which is consequently omitted here.

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 or 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 sequence arguments in Relation operator application predicates and Function or generic operator application expressions.

d) From Syntactic transformation rules:

   1) `Name` and `ExpressionList`.

e) From Type inference rules:

   1) Properties of the type inference system.

f) From Instantiation:

   1) Carrier set and Generic type instantiation.

g) From Semantic relations:

1) all of 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

$$\frac{\begin{array}{l}\{\} \vdash^{\mathcal{S}} s_{prelude} \fatsemi \Gamma_0 \\ \{prelude \mapsto \Gamma_0\} \vdash^{\mathcal{S}} s_{\rho(1)} \fatsemi \Gamma_{\rho(1)} \\ \ldots \\ \{prelude \mapsto \Gamma_0,\ i_{\rho(1)} \mapsto \Gamma_{\rho(1)},\ \ldots,\ i_{\rho(n-1)} \mapsto \Gamma_{\rho(n-1)}\} \vdash^{\mathcal{S}} s_{\rho(n)} \fatsemi \Gamma_{\rho(n)}\end{array}}{\vdash^{\mathcal{Z}} s_1 \ldots s_n}(\rho \in 1 \mathinner{..} n \rightarrowtail\!\!\!\!\!\twoheadrightarrow 1 \mathinner{..} n)$$

The sections of a specification can be presented in any order. For a specification to be well-typed, there shall exist a bijection ρ specifying a permutation of the sections so that each section is well-typed in the corresponding section-type environment. The parents relation constrains the permutations that produce a well-typed specification.

The prelude is specified as being included in the environment first. However, when typechecking $\vdash^{\mathcal{Z}} s_{prelude}$, the prelude shall be omitted from the environment.

C.2.2.3 Semantics

$$[\![\ s_1\ \ldots\ s_n\ ]\!]^{\mathcal{Z}} \quad = \quad ([\![\ \text{section}\ prelude\ldots\ ]\!]^{\mathcal{S}} \fatsemi [\![\ s_1\ ]\!]^{\mathcal{S}} \fatsemi \ldots \fatsemi [\![\ s_n\ ]\!]^{\mathcal{S}})\ \varnothing$$

The meaning of the Z specification $s_1\ \ldots\ s_n$ is the set of named theories to which the empty set of named theories is related by the composition of the relations between sets of named theories that denote the meaning of each section, starting with the prelude.

To determine $[\![\ \text{section}\ prelude\ldots\ ]\!]^{\mathcal{Z}}$ another prelude shall not be prefixed onto it.

NOTE 1 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 that has a name — shown here as *Specification* — and whose parents are (implicitly *prelude* and) *standard_toolkit*.

$$d_1 \ ... \ d_n \implies \text{section } \mathit{Specification} \text{ parents } \mathit{standard\_toolkit} \ \texttt{END} \ d_1 \ ... \ d_n$$

The name given to the section is not important: it need not be *Specification*, though it may not be *prelude* or that of any section of the mathematical toolkit.

NOTE 1 If the section is contained in a file, then the name of that file might be a good choice.

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 = section , NAME , parents , [ NAME , { ,-tok , NAME } ] , END , { Paragraph }
        | ...
        ;
```

C.3.2.2 Type

$$\frac{\Sigma_0 \vdash^{\mathcal{D}} d_1 \fatsemi [\sigma_1] \quad ... \quad \Sigma_{n-1} \vdash^{\mathcal{D}} d_n \fatsemi [\sigma_n] \quad \Lambda \vdash^{\mathcal{S}} s}{\begin{array}{c}\Lambda \vdash^{\mathcal{S}} \text{section } i \text{ parents } i_1, ..., i_m \ \texttt{END} \\ d_1 \fatsemi [\sigma_1] \ ... \ d_n \fatsemi [\sigma_n] \fatsemi \Gamma\end{array}} \left(\begin{array}{l} i \notin dom\ \Lambda \\ \{i_1, \ ..., \ i_m\} \subseteq dom\ \Lambda \\ dom\ \sigma_1 \cap dom\ \sigma_2 = \varnothing \land ... \land dom\ \sigma_1 \cap dom\ \sigma_n = \varnothing \\ \land \vdots \\ \land\ dom\ \sigma_{n-1} \cap dom\ \sigma_n = \varnothing \\ \Gamma \in (\_ \pfun \_) \end{array} \right)$$

where $\Gamma_{-1} = \text{if } i = \mathit{prelude} \text{ then } \{\} \text{ else } \Lambda\ \mathit{prelude}$
and $\Gamma_0 = \Gamma_{-1} \cup \Lambda\ i_1 \cup ... \cup \Lambda\ i_m$
and $\Gamma = \Gamma_0 \cup \{j : \texttt{NAME}; \ \tau : \texttt{Type} \mid j \mapsto \tau \in \sigma_1 \cup ... \cup \sigma_n \bullet j \mapsto (i, \ \tau)\}$
and $\Sigma_0 = \Gamma_0 \fatsemi second$
and $\Sigma_1 = \Sigma_0 \cup \sigma_1$ *and ... and* $\Sigma_{n-1} = \Sigma_{n-2} \cup \sigma_{n-1}$
and $s = \text{section } i \text{ parents } i_1, ..., i_m \ \texttt{END} \ d_1 \ ... \ d_{n-1}$

Each paragraph of an inheriting section is typechecked in an environment formed from those of the parent sections extended with the signatures of the preceding paragraphs of this section. A further type subsequent checks that the section that remains when the last paragraph is omitted is also well-typed. If the section has no paragraphs, no such type subsequent shall be generated.

NOTE 1 In other words, separate well-typedness conditions are checked for each paragraph-sized prefix of each section. This ensures that the instantiations of references to generics are fully determined before the definition containing those references is used in subsequent paragraphs, and so excludes examples such as the following.

EXAMPLE 1

$$empty == \varnothing$$

$$inst == empty \cup \{1, 2\}$$

This apparent inefficiency can be avoided in a tool implementation — see 13.3.5.

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

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

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

c) there is no global redefinition between any pair of paragraphs of the section (specified by an enumeration of pairwise disjointness constraints);

d) 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 partial function).

NOTE 2 Ancestors need not be immediate parents, and a section cannot be amongst its own ancestors (no cycles in the parent relation).

NOTE 3 The name of a section can be the same as the name of a declaration — the two are not confused.

C.3.2.3 Semantics

NOTE 1 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.

$$[\![\text{ section } prelude \text{ parents } \texttt{END}\ d_1 \ldots d_n \ ]\!]^{\mathcal{S}} = \lambda\ T : Theory \bullet \{prelude \mapsto ([\![d_1]\!]^{\mathcal{D}} \fatsemi \ldots \fatsemi [\![d_n]\!]^{\mathcal{D}})\ (\!|\ \{\varnothing\}\ |\!)\}$$

The meaning of the prelude section is given by that constant function which, whatever set of named theories it is given, returns the singleton set containing the theory named prelude, whose models are those to which the empty set of models is related by the composition of the relations between models that denote the meanings of each of its paragraphs – see clause 11 for details of those paragraphs.

$$[\![\text{ section } i \text{ parents } i_1, \ldots, i_m \ \texttt{END}\ d_1 \ldots d_n \ ]\!]^{\mathcal{S}} = \lambda\ T : Theory \bullet T \cup \{i \mapsto ([\![d_1]\!]^{\mathcal{D}} \fatsemi \ldots \fatsemi [\![d_n]\!]^{\mathcal{D}})\ (\!|\ \{M_0 : T\ prelude;\ M_1 : T\ i_1;\ \ldots;\ M_m : T\ i_m \bullet M_0 \cup M_1 \cup \ldots \cup M_m\}\ |\!)\}$$

The meaning of a section other than the prelude is the function that augments a given set of named theories with the named theory of the given section. The models in that named theory are those 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          =  ...
                 |  section , NAME , END , { Paragraph }
                 ;
```

C.3.3.2 Transformation

The base section section i `END` d_1 ... d_n is semantically equivalent to the inheriting section that inherits from no parents (bar *prelude*).

$$\text{section } i \texttt{ END } d_1 \ldots d_n \implies \text{section } i \text{ parents } \texttt{END } d_1 \ldots d_n$$

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 type, conjecture, generic conjecture, or operator template.

C.4.2 Given types

C.4.2.1 Syntax

```
Paragraph        =  [-tok , NAME , { ,-tok , NAME } , ]-tok , END
                 |  ...
                 ;
```

C.4.2.2 Type

$$\frac{}{\Sigma \vdash^{\mathcal{D}} [i_1, ..., i_n] \texttt{ END} \mathrel{\mathfrak{s}} [i_1 : \mathbb{P}(\texttt{GIVEN } i_1); \ ...; \ i_n : \mathbb{P}(\texttt{GIVEN } i_n)]} (\# \{i_1, \ ..., \ i_n\} = n)$$

In a given types paragraph, the annotation of the paragraph is a signature associating the given type names with set types. There shall be no duplication of names within a given types paragraph.

C.4.2.3 Semantics

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

$$\begin{aligned} [\![\, [i_1, ..., i_n] \texttt{ END} \,]\!]^{\mathcal{D}} \quad = \quad & \{M : \mathit{Model}; \ w_1, \ ..., \ w_n : \mathbb{W} \\ & \bullet \ M \mapsto M \cup \{i_1 \mapsto w_1, \ ..., \ i_n \mapsto w_n\} \\ & \qquad \cup \{\mathit{decor} \heartsuit i_1 \mapsto w_1, \ ..., \ \mathit{decor} \heartsuit i_n \mapsto w_n\}\} \end{aligned}$$

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 (16.2.6.1) can avoid being captured.

C.4.3 Axiomatic description

C.4.3.1 Syntax

```
Paragraph = ...
          | AX , SchemaText , END
          | ...
          ;
```

C.4.3.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{D}} \texttt{AX}\ e\ \texttt{END} \mathbin{:} [\sigma]}$$

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.

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

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.

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.

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

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.

C.4.5 Generic axiomatic description

C.4.5.1 Syntax

```
Paragraph = ...
          | GENAX , [-tok , Formals , ]-tok , SchemaText , END
          | ...
          ;
```

C.4.5.2 Type

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_1),\ ...,\ i_n \mapsto \mathbb{P}(\texttt{GENTYPE}\ i_n)\} \vdash^{\mathcal{E}} e \ \boldsymbol{:}\ \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{D}} \texttt{GENAX}\ [i_1, ..., i_n]\ e\ \texttt{END} \ \boldsymbol{:}\ [\lambda\ j : dom\ \sigma \bullet [i_1, ..., i_n](\sigma\ j)]} (\#\ \{i_1,\ ...,\ i_n\} = n)$$

In a generic axiomatic description paragraph $\texttt{GENAX}\ [i_1, ..., i_n]\ e\ \texttt{END}$, 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. There shall be no duplication of names within the generic parameters of a generic axiomatic description paragraph.

C.4.5.3 Semantics

The generic axiomatic description paragraph $\texttt{GENAX}\ [i_1, ..., i_n]\ e\ \texttt{END}$ introduces global names and constraints on their values, with generic parameters that have to be instantiated (by sets) whenever those names are referenced.

$$\begin{aligned}
&[\![\ \texttt{GENAX}\ [i_1, ..., i_n]\ (e\ \boldsymbol{:}\ \mathbb{P}[j_1 : \tau_1;\ ...;\ j_m : \tau_m])\ \texttt{END}\ ]\!]^{\mathcal{D}} = \\
&\quad \{M : Model;\ u : \mathbb{W} \to \mathbb{W} \\
&\qquad \mid \forall\ w_1,\ ...,\ w_n : \mathbb{W} \bullet \exists\ w : \mathbb{W} \bullet \\
&\qquad\quad u\ (w_1,\ ...,\ w_n) \in w \\
&\qquad\quad \wedge\ (M \oplus \{i_1 \mapsto w_1,\ ...,\ i_n \mapsto w_n\} \cup \{decor\ \spadesuit\ i_1 \mapsto w_1,\ ...,\ decor\ \spadesuit\ i_n \mapsto w_n\}) \mapsto w \in [\![\ e\ ]\!]^{\mathcal{E}} \\
&\qquad \bullet\ M \mapsto M \cup \lambda\ y : \{j_1,\ ...,\ j_m\} \bullet \lambda\ x : dom\ u \bullet u\ x\ y\}
\end{aligned}$$

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 (16.2.6.2) 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 $\texttt{GENSCH}\ i\ [i_1, ..., i_n]\ t\ \texttt{END}$ 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        =  ...
                 |  DeclName , == , Expression , END
                 |  ...
                 ;
```

C.4.7.2 Transformation

The horizontal definition paragraph i == e END introduces the global name i, associating with it the value of e.

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

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        =  ...
                 |  NAME , [-tok , Formals , ]-tok , == , Expression , END
                 |  ...
                 ;
```

C.4.8.2 Transformation

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

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

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        =  ...
                 |  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 `NAME`s 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.

C.4.9.3 `PrefixGenName`

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

C.4.9.4 `PostfixGenName`

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

C.4.9.5 `InfixGenName`

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

C.4.9.6 `NofixGenName`

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

C.4.10 Free type

C.4.10.1 Syntax

```
Paragraph  = ...
           | Freetype , { & , Freetype } , END
           | ...
           ;

Freetype   = NAME , ::= , Branch , { |-tok , Branch } ;

Branch     = DeclName , [ ⟪ , Expression , ⟫ ] ;
```

C.4.10.2 Transformation

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

$$\ldots \mid g\langle\langle e\rangle\rangle \mid h \mid \ldots \quad \Longrightarrow \quad \ldots \mid h \mid g\langle\langle e\rangle\rangle \mid \ldots$$

Exhaustive application of this syntactic transformation rule effects a sort.

The second stage requires implicit generic instantiation expressions to have been filled in, which is done during typechecking. Hence that second stage is delayed until after typechecking, where it appears in the form of a semantic transformation rule (section 15.2.3).

C.4.10.3 Type

$$\frac{\begin{array}{lll} \Sigma_0 \vdash^{\mathcal{E}} e_{1\,1} \mathbin{:} \mathbb{P}\,\tau_{1\,1} & \ldots & \Sigma_0 \vdash^{\mathcal{E}} e_{1\,n_1} \mathbin{:} \mathbb{P}\,\tau_{1\,n_1} \\ \vdots & & \\ \Sigma_0 \vdash^{\mathcal{E}} e_{r\,1} \mathbin{:} \mathbb{P}\,\tau_{r\,1} & \ldots & \Sigma_0 \vdash^{\mathcal{E}} e_{r\,n_r} \mathbin{:} \mathbb{P}\,\tau_{r\,n_r} \end{array}}{\Sigma \vdash^{\mathcal{D}} d \mathbin{:} [\sigma]} \left(\begin{array}{l} \#\{f_1,\, h_{1\,1},\, \ldots,\, h_{1\,m_1},\, g_{1\,1},\, \ldots,\, g_{1\,n_1}, \\ \qquad \vdots, \\ \qquad f_r,\, h_{r\,1},\, \ldots,\, h_{r\,m_r},\, g_{r\,1},\, \ldots,\, g_{r\,n_r}\} \\ = r + m_1 + \ldots + m_r + n_1 + \ldots + n_r \end{array} \right)$$

$$\begin{array}{lll} \textit{where } \Sigma_0 = \Sigma \oplus \{f_1 \mapsto \mathbb{P}\,f_1,\, \ldots,\, f_r \mapsto \mathbb{P}\,f_r\} & & \\ \quad \textit{and } d & = & f_1 ::= h_{1\,1} \mid \ldots \mid h_{1\,m_1} \mid g_{1\,1}\langle\langle e_{1\,1}\rangle\rangle \mid \ldots \mid g_{1\,n_1}\langle\langle e_{1\,n_1}\rangle\rangle \\ & & \&\ \ldots\ \& \\ & & f_r ::= h_{r\,1} \mid \ldots \mid h_{r\,m_r} \mid g_{r\,1}\langle\langle e_{r\,1}\rangle\rangle \mid \ldots \mid g_{r\,n_r}\langle\langle e_{r\,n_r}\rangle\rangle \ \texttt{END} \\ \quad \textit{and } \sigma & = & f_1 : \mathbb{P}\,f_1;\ h_{1\,1} : f_1;\ \ldots;\ h_{1\,m_1} : f_1;\ g_{1\,1} : \mathbb{P}(\tau_{1\,1} \times f_1);\ \ldots;\ g_{1\,n_1} : \mathbb{P}(\tau_{1\,n_1} \times f_1) \\ & & ;\ \ldots; \\ & & f_r : \mathbb{P}\,f_r;\ h_{r\,1} : f_r;\ \ldots;\ h_{r\,m_r} : f_r;\ g_{r\,1} : \mathbb{P}(\tau_{r\,1} \times f_r);\ \ldots;\ g_{r\,n_r} : \mathbb{P}(\tau_{r\,n_r} \times f_r) \end{array}$$

In a free type paragraph d, as expanded in the second local definition, 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 relevant type. There shall be no duplication of names amongst the free types, elements and injections of a free type paragraph.

C.4.10.4 Semantics

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

$$\begin{array}{l} f_1 ::= h_{1\,1} \mid \ldots \mid h_{1\,m_1} \mid g_{1\,1}\langle\langle e_{1\,1}\rangle\rangle \mid \ldots \mid g_{1\,n_1}\langle\langle e_{1\,n_1}\rangle\rangle \\ \&\ \ldots\ \& \\ f_r ::= h_{r\,1} \mid \ldots \mid h_{r\,m_r} \mid g_{r\,1}\langle\langle e_{r\,1}\rangle\rangle \mid \ldots \mid g_{r\,n_r}\langle\langle e_{r\,n_r}\rangle\rangle \end{array}$$

$$\Longrightarrow$$

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

AX
$h_{1\,1},\ ...,\ h_{1\,m_1} : f_1$
$\vdots$
$h_{r\,1},\ ...,\ h_{r\,m_r} : f_r$
$g_{1\,1} : \mathbb{P}(e_{1\,1} \times f_1);\ ...;\ g_{1\,n_1} : \mathbb{P}(e_{1\,n_1} \times f_1)$
$\vdots$
$g_{r\,1} : \mathbb{P}(e_{r\,1} \times f_r);\ ...;\ g_{r\,n_r} : \mathbb{P}(e_{r\,n_r} \times f_r)$
$|$
$\forall\ u : e_{1\,1} \bullet \exists_1\ x : g_{1\,1} \bullet x\ .\ 1 = u \land ... \land \forall\ u : e_{1\,n_1} \bullet \exists_1\ x : g_{1\,n_1} \bullet x\ .\ 1 = u$
$\vdots \land$
$\forall\ u : e_{r\,1} \bullet \exists_1\ x : g_{r\,1} \bullet x\ .\ 1 = u \land ... \land \forall\ u : e_{r\,n_r} \bullet \exists_1\ x : g_{r\,n_r} \bullet x\ .\ 1 = u$

$\forall\ u, v : e_{1\,1} \mid g_{1\,1} u = g_{1\,1} v \bullet u = v \land ... \land \forall\ u, v : e_{1\,n_1} \mid g_{1\,n_1} u = g_{1\,n_1} v \bullet u = v$
$\vdots \land$
$\forall\ u, v : e_{r\,1} \mid g_{r\,1} u = g_{r\,1} v \bullet u = v \land ... \land \forall\ u, v : e_{r\,n_r} \mid g_{r\,n_r} u = g_{r\,n_r} v \bullet u = v$

$\forall\ b_1, b_2 : \mathbb{N} \bullet$
$(\forall\ w : f_1 \mid$
$(b_1 = 1 \land w = h_{1\,1} \lor ... \lor b_1 = m_1 \land w = h_{1\,m_1} \lor$
$b_1 = m_1 + 1 \land w \in \{x : g_{1\,1} \bullet x\ .\ 2\} \lor ... \lor b_1 = m_1 + n_1 \land w \in \{x : g_{1\,n_1} \bullet x\ .\ 2\})$
$\land\ (b_2 = 1 \land w = h_{1\,1} \lor ... \lor b_2 = m_1 \land w = h_{1\,m_1} \lor$
$b_2 = m_1 + 1 \land w \in \{x : g_{1\,1} \bullet x\ .\ 2\} \lor ... \lor b_2 = m_1 + n_1 \land w \in \{x : g_{1\,n_1} \bullet x\ .\ 2\}) \bullet$
$b_1 = b_2) \land$
$\vdots \land$
$(\forall\ w : f_r \mid$
$(b_1 = 1 \land w = h_{r\,1} \lor ... \lor b_1 = m_r \land w = h_{r\,m_r} \lor$
$b_1 = m_r + 1 \land w \in \{x : g_{r\,1} \bullet x\ .\ 2\} \lor ... \lor b_1 = m_r + n_r \land w \in \{x : g_{r\,n_r} \bullet x\ .\ 2\})$
$\land\ (b_2 = 1 \land w = h_{r\,1} \lor ... \lor b_2 = m_r \land w = h_{r\,m_r} \lor$
$b_2 = m_r + 1 \land w \in \{x : g_{r\,1} \bullet x\ .\ 2\} \lor ... \lor b_2 = m_r + n_r \land w \in \{x : g_{r\,n_r} \bullet x\ .\ 2\}) \bullet$
$b_1 = b_2)$

$\forall\ w_1 : \mathbb{P}\, f_1;\ ...;\ w_r : \mathbb{P}\, f_r \mid$
$h_{1\,1} \in w_1 \land ... \land h_{1\,m_1} \in w_1 \land$
$\vdots \land$
$h_{r\,1} \in w_r \land ... \land h_{r\,m_r} \in w_r \land$
$(\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{1\,1}) \bullet g_{1\,1} y \in w_1) \land$
$... \land (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{1\,n_1}) \bullet g_{1\,n_1} y \in w_1) \land$
$\vdots \land$
$(\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{r\,1}) \bullet g_{r\,1} y \in w_r) \land$
$... \land (\forall\ y : (\mu\ f_1 == w_1;\ ...;\ f_r == w_r \bullet e_{r\,n_r}) \bullet g_{r\,n_r} y \in w_r) \bullet$
$w_1 = f_1 \land ... \land w_r = f_r$
END

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 1 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 2 The right-hand side of this transformation could have been expressed using the following notation from the mathematical toolkit, but for the desire to define the core language separately from the mathematical toolkit.

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

C.4.11 Conjecture

C.4.11.1 Syntax

```
Paragraph = ...
          | |=? , Predicate , END
          | ...
          ;
```

C.4.11.2 Type

$$\frac{\Sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} \models? \; p \; \texttt{END} \; \mathbin{:} [\,]}$$

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

C.4.11.3 Semantics

The conjecture paragraph $\models? \; p \; \texttt{END}$ expresses a property that may logically follow from the specification. It may be a starting point for a proof.

$$[\![\models? \; p \; \texttt{END}]\!]^{\mathcal{D}} = id \; Model$$

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 = ...
          | [-tok , Formals , ]-tok , |=? , Predicate , END
          | ...
          ;
```

C.4.12.2 Type

$$\frac{\Sigma \oplus \{i_1 \mapsto \mathbb{P}(\texttt{GENTYPE} \; i_1), \; ..., \; i_n \mapsto \mathbb{P}(\texttt{GENTYPE} \; i_n)\} \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{D}} [i_1, ..., i_n] \models? \; p \; \texttt{END} \; \mathbin{:} [\,]} (\# \{i_1, \; ..., \; i_n\} = n)$$

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

C.4.12.3 Semantics

The generic conjecture paragraph $[i_1, ..., i_n] \models? \; p \; \texttt{END}$ expresses a generic property that may logically follow from the specification. It may be a starting point for a proof.

$$[\![[i_1, ..., i_n] \models? \; p \; \texttt{END}]\!]^{\mathcal{D}} = id \; Model$$

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 subsequent occurrences in this section of the words in the template, with whatever strokes they are decorated, as particular prefix, infix, postfix or nofix names.

C.4.13.1 Syntax

```
Paragraph         = ...
                  | OperatorTemplate , END
                  ;

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

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

Prec              = NUMBER ;

Assoc             = leftassoc
                  | rightassoc
                  ;

Template          = PrefixTemplate
                  | PostfixTemplate
                  | InfixTemplate
                  | NofixTemplate
                  ;

PrefixTemplate    = (-tok , NAME , { ( _ | ,, ) , NAME } , _ , )-tok ;

PostfixTemplate   = (-tok , _ , NAME , { ( _ | ,, ) , NAME } , )-tok ;

InfixTemplate     = (-tok , _ , NAME , { ( _ | ,, ) , NAME } , _ , )-tok ;

NofixTemplate     = (-tok , NAME , { ( _ | ,, ) , NAME } , )-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

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{⦂} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \forall\, e \bullet p}$$

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.

$$[\![\, \forall\, e \bullet p \,]\!]^{\mathcal{P}} = \{M : Model \mid \forall\, t : [\![\, e \,]\!]^{\mathcal{E}} M \bullet M \oplus t \in [\![\, p \,]\!]^{\mathcal{P}} \bullet M\}$$

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.

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

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

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{⦂} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\mathcal{P}} \exists_1\, e \bullet p}$$

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*.

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

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*.

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

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*.

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

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 $p_1 \lor p_2$ is *true* if and only if at least one of p_1 and p_2 is *true*.

$$p_1 \lor p_2 \quad \Longrightarrow \quad \neg\, (\neg\, p_1 \land \neg\, p_2)$$

It is semantically equivalent to not both of them being *false*.

C.5.10 Conjunction

C.5.10.1 Syntax

```
Predicate = ...
          | Predicate , ∧ , Predicate
          | ...
          ;
```

C.5.10.2 Type

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

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

C.5.10.3 Semantics

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.

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*.

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

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.

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 rule 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.5.12.3 `PrefixRel`

$$
\begin{aligned}
prep\ e &\implies e \in prep\bowtie \\
lp\ e_1\ es\ \ldots\ e_{n-2}\ es\ e_{n-1}\ erep\ e_n &\implies (e_1, \ldots, e_{n-2}, e_{n-1}, e_n) \in lp\bowtie es \ldots \bowtie es \bowtie erep\bowtie \\
lp\ e_1\ es\ \ldots\ e_{n-2}\ es\ se_{n-1}\ srep\ e_n &\implies (e_1, \ldots, e_{n-2}, se_{n-1}, e_n) \in lp\bowtie es \ldots \bowtie es \bowtie srep\bowtie
\end{aligned}
$$

C.5.12.4 `PostfixRel`

$$
\begin{aligned}
e\ postp &\implies e \in \bowtie postp \\
e_1\ elp\ e_2\ es\ \ldots\ e_{n-1}\ es\ e_n\ erp &\implies (e_1, e_2, \ldots, e_{n-1}, e_n) \in \bowtie elp \bowtie es \ldots \bowtie es \bowtie erp \\
e_1\ elp\ e_2\ es\ \ldots\ e_{n-1}\ es\ se_n\ srp &\implies (e_1, e_2, \ldots, e_{n-1}, se_n) \in \bowtie elp \bowtie es \ldots \bowtie es \bowtie srp
\end{aligned}
$$

C.5.12.5 `InfixRel`

$$e_1\ ip_1\ e_2\ ip_2\ e_3\ \ldots \implies e_1\ ip_1\ e_2 \mathbin{:} \tau_1 \land e_2 \mathbin{:} \tau_1\ ip_2\ e_3 \mathbin{:} \tau_2\ \ldots$$

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

$$
\begin{aligned}
e_1 = e_2 &\implies e_1 \in \{e_2\} \\
e_1\ ip\ e_2 &\implies (e_1, e_2) \in (\_\ ip\ \_)
\end{aligned}
$$

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

$$e_1\ elp\ e_2\ es\ ...\ e_{n-2}\ es\ e_{n-1}\ erep\ e_n \implies (e_1, e_2, ..., e_{n-2}, e_{n-1}, e_n) \in \bowtie elp \bowtie es ... \bowtie es \bowtie erep \bowtie$$
$$e_1\ elp\ e_2\ es\ ...\ e_{n-2}\ es\ se_{n-1}\ srep\ e_n \implies (e_1, e_2, ..., e_{n-2}, se_{n-1}, e_n) \in \bowtie elp \bowtie es ... \bowtie es \bowtie srep \bowtie$$

C.5.12.6 `NofixRel`

$$lp\ e_1\ es\ ...\ e_{n-1}\ es\ e_n\ erp \implies (e_1, ..., e_{n-1}, e_n) \in lp \bowtie es ... \bowtie es \bowtie erp$$
$$lp\ e_1\ es\ ...\ e_{n-1}\ es\ se_n\ srp \implies (e_1, ..., e_{n-1}, se_n) \in lp \bowtie es ... \bowtie es \bowtie srp$$

C.5.12.7 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \ : \ \tau \qquad \Sigma \vdash^{\mathcal{E}} e_2 \ : \ \mathbb{P}\tau}{\Sigma \vdash^{\mathcal{P}} e_1 \in e_2}$$

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.5.12.8 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.

$$[\![e_1 \in e_2]\!]^{\mathcal{P}} = \{M : Model \mid [\![e_1]\!]^{\mathcal{E}} M \in [\![e_2]\!]^{\mathcal{E}} M \bullet M\}$$

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 or 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

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{:} \mathbb{P}[\sigma_1] \qquad \Sigma \oplus \sigma_1 \vdash^{\mathcal{E}} e_2 \mathbin{:} \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} \forall\, e_1 \bullet e_2 \mathbin{:} \mathbb{P}[dom\ \sigma_1 \lhd \sigma_2]}(\sigma_1 \approx \sigma_2)$$

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.

$$[\![\ \forall\, e_1 \bullet e_2 \mathbin{:} \mathbb{P}\,\tau\ ]\!]^{\mathcal{E}} \;=\; \lambda\, M : Model \bullet \{t_2 : [\![\ \tau\ ]\!]^{\mathcal{T}} M \mid \forall\, t_1 : [\![\ e_1\ ]\!]^{\mathcal{E}} M \bullet t_1 \cup t_2 \in [\![\ e_2\ ]\!]^{\mathcal{E}}\ (M \oplus t_1) \bullet t_2\}$$

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.

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

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \ : \ \mathbb{P}[\sigma_1] \qquad \Sigma \oplus \sigma_1 \vdash^{\varepsilon} e_2 \ : \ \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\varepsilon} \exists_1\ e_1 \bullet e_2 \ : \ \mathbb{P}[dom\ \sigma_1 \lhd \sigma_2]}(\sigma_1 \approx \sigma_2)$$

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.

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

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

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\varepsilon} e_2 \mathbin{:} \tau}{\Sigma \vdash^{\varepsilon} \mu\ e_1 \bullet e_2 \mathbin{:} \tau}$$

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 unique value of e_2 that arises whichever binding is chosen from the set that is the value of schema e_1.

$$\{M : Model;\ t_1 : \mathbb{W} \mid t_1 \in [\![\ e_1\ ]\!]^{\varepsilon} M \wedge (\forall\ t_3 : [\![\ e_1\ ]\!]^{\varepsilon} M \bullet [\![\ e_2\ ]\!]^{\varepsilon}\ (M \oplus t_3) = [\![\ e_2\ ]\!]^{\varepsilon}\ (M \oplus t_1)) \bullet M \mapsto [\![\ e_2\ ]\!]^{\varepsilon}\ (M \oplus t_1)\} \subseteq [\![\ \mu\ e_1 \bullet e_2\ ]\!]^{\varepsilon}$$

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

$$\mathrm{let}\ i_1 == e_1;\ ...;\ i_n == e_n \bullet e \implies \mu\ i_1 == e_1;\ ...;\ i_n == i_n \bullet e$$

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.

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

It is semantically equivalent to a schema conjunction.

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.

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

It is semantically equivalent to a schema disjunction.

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 $e_1 \lor e_2$ 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.

$$e_1 \lor e_2 \quad \Longrightarrow \quad \neg\, (\neg\, e_1 \land \neg\, e_2)$$

It is semantically equivalent to a schema negation.

C.6.11 Schema conjunction

C.6.11.1 Syntax

```
Expression      =  ...
                |  Expression , ∧ , Expression
                |  ...
                ;
```

C.6.11.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{:} \mathbb{P}[\sigma_1] \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathbin{:} \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} e_1 \wedge e_2 \mathbin{:} \mathbb{P}[\sigma_1 \cup \sigma_2]} (\sigma_1 \approx \sigma_2)$$

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.

$$[\![e_1 \wedge e_2 \mathbin{:} \mathbb{P}\, \tau]\!]^{\mathcal{E}} = \lambda\, M : Model \bullet \{ t : [\![\tau]\!]^{\mathcal{T}} M;\ t_1 : [\![e_1]\!]^{\mathcal{E}} M;\ t_2 : [\![e_2]\!]^{\mathcal{E}} M \mid t_1 \cup t_2 = t \bullet t \}$$

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.

C.6.12 Schema negation

C.6.12.1 Syntax

```
Expression      =  ...
                |  ¬ , Expression
                |  ...
                ;
```

C.6.12.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e \mathbin{:} \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{E}} \neg\, e \mathbin{:} \mathbb{P}[\sigma]}$$

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 which 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.

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*.

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

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

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \mathbin{⦂} \mathbb{P}[\sigma_1] \qquad \Sigma \vdash^{\mathcal{E}} e_2 \mathbin{⦂} \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} e_1 \fatsemi e_2 \mathbin{⦂} \mathbb{P}[\sigma_3 \cup \sigma_4]} \left(\begin{array}{l} \sigma_3 \approx \sigma_4 \\ \{i : match \bullet i \mapsto \sigma_1(decor\ '\ i)\} \approx \{i : match \bullet i \mapsto \sigma_2 i\} \end{array} \right)$$

where $match = \{\, i : dom\ \sigma_2 \mid decor\ '\ i \in dom\ \sigma_1 \land (\forall\, j : \mathtt{NAME} \bullet \neg\, i = decor\ '\ j) \bullet i\}$
and $\sigma_3 = \{i : match \bullet decor\ '\ i\} \lhd \sigma_1$
and $\sigma_4 = match \lhd \sigma_2$

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 unprimed names in schema e_2 for which there are matching primed names in schema e_1. Let σ_3 be the signature formed from the components of e_1 excluding the matched primed components. Let σ_4 be the signature formed from the components of e_2 excluding the matched unprimed components. Signatures σ_3 and σ_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 σ_3 and σ_4.

NOTE 1 This notation would not be associative without the restriction concerning names being unprimed.

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.

$$(e_1 \colon \mathbb{P}[\sigma_1]) \fatsemi (e_2 \colon \mathbb{P}[\sigma_2]) \colon \mathbb{P}[\sigma] \implies \begin{array}{l} \mathrm{let}\, e_3 == \mathit{carrier}\ [\sigma_1 \setminus \sigma]; \\ \quad e_4 == \mathit{carrier}\ [\sigma_2 \setminus \sigma] \\ \quad \bullet\ \mathrm{let}\, e^{\bowtie} == e_4 \mathit{uniquely\ renamed} \\ \quad\quad \bullet\ \exists\ e^{\bowtie} \bullet (\exists\ e_3 \bullet [e_1;\ e^{\bowtie} \mid \theta\, e_3 = \theta\, e^{\bowtie}]) \\ \quad\quad\quad \land\ (\exists\ e_4 \bullet [e_2;\ e^{\bowtie} \mid \theta\, e_4 = \theta\, e^{\bowtie}]) \end{array}$$

It is semantically equivalent to the existential quantification of the matched pairs of primed components of e_1 and unprimed components of e_2 (as given by the signatures determined by typechecking), with those matched pairs being equated.

C.6.15 Schema piping

C.6.15.1 Syntax

```
Expression = ...
           | Expression , ≫ , Expression
           | ...
           ;
```

C.6.15.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e_1 \colon \mathbb{P}[\sigma_1] \qquad \Sigma \vdash^{\mathcal{E}} e_2 \colon \mathbb{P}[\sigma_2]}{\Sigma \vdash^{\mathcal{E}} e_1 \gg e_2 \colon \mathbb{P}[\sigma_3 \cup \sigma_4]} \left(\begin{array}{l} \sigma_3 \approx \sigma_4 \\ \{i : \mathit{match} \bullet i \mapsto \sigma_1(\mathit{decor}\ !\ i)\} \approx \{i : \mathit{match} \bullet i \mapsto \sigma_2(\mathit{decor}\ ?\ i)\} \end{array} \right)$$

$$\begin{array}{l} \mathit{where}\ \mathit{match} = \{\, i : \mathtt{NAME} \mid \mathit{decor}\ !\ i \in \mathit{dom}\ \sigma_1 \land \mathit{decor}\ ?\ i \in \mathit{dom}\ \sigma_2 \land (\forall\ j : \mathtt{NAME} \bullet \neg\ i = \mathit{decor}\ !\ j) \bullet i\} \\ \quad \mathit{and}\ \sigma_3 = \{i : \mathit{match} \bullet \mathit{decor}\ !\ i\} \lhd \sigma_1 \\ \quad \mathit{and}\ \sigma_4 = \{i : \mathit{match} \bullet \mathit{decor}\ ?\ i\} \lhd \sigma_2 \end{array}$$

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 unshrieked names for which there are shrieked names in schema e_1 matching queried names in schema e_2. Let σ_3 be the signature formed from the components of e_1 excluding the matched shrieked components. Let σ_4 be the signature formed from the components of e_2 excluding the matched queried components. Signatures σ_3 and σ_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 σ_3 and σ_4.

NOTE 1 This notation would not be associative without the restriction concerning names being unshrieked.

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.

$$(e_1 \colon \mathbb{P}[\sigma_1]) \gg (e_2 \colon \mathbb{P}[\sigma_2]) \colon \mathbb{P}[\sigma] \implies \begin{array}{l} \mathrm{let}\, e_3 == \mathit{carrier}\ [\sigma_1 \setminus \sigma]; \\ \quad e_4 == \mathit{carrier}\ [\sigma_2 \setminus \sigma] \\ \quad \bullet\ \mathrm{let}\, e^{\bowtie} == e_4 \mathit{uniquely\ renamed} \\ \quad\quad \bullet\ \exists\ e^{\bowtie} \bullet (\exists\ e_3 \bullet [e_1;\ e^{\bowtie} \mid \theta\, e_3 = \theta\, e^{\bowtie}]) \\ \quad\quad\quad \land\ (\exists\ e_4 \bullet [e_2;\ e^{\bowtie} \mid \theta\, e_4 = \theta\, e^{\bowtie}]) \end{array}$$

It is semantically equivalent to the existential quantification of the matched pairs of shrieked components of e_1 and queried components of e_2 (as given by the signatures determined by typechecking), 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

$$\frac{\Sigma \vdash^{\varepsilon} e \mathrel{:} \mathbb{P}[\sigma]}{\Sigma \vdash^{\varepsilon} e \setminus (i_1, ..., i_n) \mathrel{:} \mathbb{P}[\{i_1, \ldots, i_n\} \lhd \sigma]} (\{i_1, \ldots, i_m\} \subseteq dom\ \sigma)$$

In a schema hiding expression $e \setminus (i_1, ..., i_n)$, expression e shall be a schema, and the names 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 to be 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.

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

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

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 $e_1 \land e_2$ but whose signature is restricted to just that of schema e_2.

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

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.

C.6.18 Schema precondition

C.6.18.1 Syntax

```
Expression = ...
           | pre , Expression
           | ...
           ;
```

C.6.18.2 Type

$$\frac{\Sigma \vdash^{\mathcal{E}} e : \mathbb{P}[\sigma]}{\Sigma \vdash^{\mathcal{E}} \mathrm{pre}\, e : \mathbb{P}[\{i, j : \mathtt{NAME} \mid j \in dom\ \sigma \land (j = decor\ '\ i \lor j = decor\ !\ i) \bullet j\} \lhd \sigma]}$$

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.

$$\mathrm{pre}(e : \mathbb{P}[\sigma_1]) : \mathbb{P}[\sigma_2] \implies \exists\ carrier\ [\sigma_1 \setminus \sigma_2] \bullet e$$

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.

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

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

C.6.20 Function or generic operator application

C.6.20.1 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.6.20.2 Transformation

All function operator applications are transformed to annotated application expressions.

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

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 rule 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.6.20.3 `PrefixApp`

$$
\begin{aligned}
pre\ e &\implies pre \bowtie e \\
ln\ e_1\ es\ \ldots\ e_{n-2}\ es\ e_{n-1}\ ere\ e_n &\implies ln \bowtie es \ldots \bowtie es \bowtie ere \bowtie\ (e_1, \ldots, e_{n-2}, e_{n-1}, e_n) \\
ln\ e_1\ es\ \ldots\ e_{n-2}\ es\ se_{n-1}\ sre\ e_n &\implies ln \bowtie es \ldots \bowtie es \bowtie sre \bowtie\ (e_1, \ldots, e_{n-2}, se_{n-1}, e_n)
\end{aligned}
$$

$$
\mathbb{P}\ e \implies \mathbb{P}\ e
$$

An application of the prefix generic operator $\mathbb{P}$ (that specific `PRE` token) is transformed to a powerset phrase of the annotated notation. Other applications of prefix generic operators are transformed to generic instantiation expressions.

$$
\begin{aligned}
pre\ e &\implies pre \bowtie\ [e] \\
ln\ e_1\ es\ \ldots\ e_{n-2}\ es\ e_{n-1}\ ere\ e_n &\implies ln \bowtie es \ldots \bowtie es \bowtie ere \bowtie\ [e_1, \ldots, e_{n-2}, e_{n-1}, e_n] \\
ln\ e_1\ es\ \ldots\ e_{n-2}\ es\ se_{n-1}\ sre\ e_n &\implies ln \bowtie es \ldots \bowtie es \bowtie sre \bowtie\ [e_1, \ldots, e_{n-2}, se_{n-1}, e_n]
\end{aligned}
$$

C.6.20.4 PostfixApp

$$
\begin{array}{rcl}
e\ post & \Longrightarrow & \bowtie post\ e \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-1}\ es\ e_n\ er & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie er\ (e_1, e_2, \ldots, e_{n-1}, e_n) \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-1}\ es\ se_n\ sr & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie sr\ (e_1, e_2, \ldots, e_{n-1}, se_n)
\end{array}
$$

$$
\begin{array}{rcl}
e\ post & \Longrightarrow & \bowtie post\ [e] \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-1}\ es\ e_n\ er & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie er\ [e_1, e_2, \ldots, e_{n-1}, e_n] \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-1}\ es\ se_n\ sr & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie sr\ [e_1, e_2, \ldots, e_{n-1}, se_n]
\end{array}
$$

C.6.20.5 InfixApp

$$
\begin{array}{rcl}
e_1\ in\ e_2 & \Longrightarrow & \bowtie in \bowtie\ (e_1, e_2) \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-2}\ es\ e_{n-1}\ ere\ e_n & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie ere \bowtie\ (e_1, e_2, \ldots, e_{n-2}, e_{n-1}, e_n) \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-2}\ es\ se_{n-1}\ sre\ e_n & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie sre \bowtie\ (e_1, e_2, \ldots, e_{n-2}, se_{n-1}, e_n)
\end{array}
$$

$$
\begin{array}{rcl}
e_1\ in\ e_2 & \Longrightarrow & \bowtie in \bowtie\ [e_1, e_2] \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-2}\ es\ e_{n-1}\ ere\ e_n & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie ere \bowtie\ [e_1, e_2, \ldots, e_{n-2}, e_{n-1}, e_n] \\
e_1\ el\ e_2\ es\ \ldots\ e_{n-2}\ es\ se_{n-1}\ sre\ e_n & \Longrightarrow & \bowtie el \bowtie es \ldots \bowtie es \bowtie sre \bowtie\ [e_1, e_2, \ldots, e_{n-2}, se_{n-1}, e_n]
\end{array}
$$

C.6.20.6 NofixApp

$$
\begin{array}{rcl}
ln\ e_1\ es\ \ldots\ e_{n-1}\ es\ e_n\ er & \Longrightarrow & ln \bowtie es \ldots \bowtie es \bowtie er\ (e_1, \ldots, e_{n-1}, e_n) \\
ln\ e_1\ es\ \ldots\ e_{n-1}\ es\ se_n\ sr & \Longrightarrow & ln \bowtie es \ldots \bowtie es \bowtie sr\ (e_1, \ldots, e_{n-1}, se_n)
\end{array}
$$

$$
\begin{array}{rcl}
ln\ e_1\ es\ \ldots\ e_{n-1}\ es\ e_n\ er & \Longrightarrow & ln \bowtie es \ldots \bowtie es \bowtie er\ [e_1, \ldots, e_{n-1}, e_n] \\
ln\ e_1\ es\ \ldots\ e_{n-1}\ es\ se_n\ sr & \Longrightarrow & ln \bowtie es \ldots \bowtie es \bowtie sr\ [e_1, \ldots, e_{n-1}, se_n]
\end{array}
$$

C.6.20.7 Type

$$\frac{\Sigma \vdash^{\varepsilon} e : \mathbb{P}\,\tau}{\Sigma \vdash^{\varepsilon} \mathbb{P}\,e : \mathbb{P}\,\mathbb{P}\,\tau}$$

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

C.6.20.8 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.6.21 Application

C.6.21.1 Syntax

```
Expression = ...
           | Expression , Expression
           | ...
           ;
```

C.6.21.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \mathbb{P}(\tau_1 \times \tau_2) \qquad \Sigma \vdash^{\varepsilon} e_2 \mathbin{:} \tau_1}{\Sigma \vdash^{\varepsilon} e_1\ e_2 \mathbin{:} \tau_2}$$

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.6.21.3 Semantics

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

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

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.

C.6.22 Schema decoration

C.6.22.1 Syntax

```
Expression = ...
           | Expression , STROKE
           | ...
           ;
```

C.6.22.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{:} \mathbb{P}[\sigma]}{\Sigma \vdash^{\varepsilon} e^{+} \mathbin{:} \mathbb{P}[\{i : dom\ \sigma \bullet decor^{+}\ i \mapsto \sigma\ i\}]}$$

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 decoration appended to each of its names.

C.6.22.3 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 $^{+}$.

$$(e \mathbin{:} \mathbb{P}[i_1 : \tau_1; \ ...; \ i_n : \tau_n])^{+} \implies e\ [decor^{+}\ i_1 \,/\, i_1, ..., decor^{+}\ i_n \,/\, i_n]$$

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

C.6.23 Schema renaming

C.6.23.1 Syntax

```
Expression      =  ...
                |  Expression , [-tok , DeclName , / , DeclName , { ,-tok , DeclName , / , DeclName } , ]-tok
                |  ...
                ;
```

C.6.23.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{⨟} \mathbb{P}[\sigma_1]}{\Sigma \vdash^{\varepsilon} e\ [j_1 \mathbin{/} i_1, ..., j_n \mathbin{/} i_n] \mathbin{⨟} \mathbb{P}[\sigma_2]} \left(\begin{array}{l} \# \{i_1, ..., i_n\} = n \\ \sigma_2 \in (\_ \pfun \_) \end{array} \right)$$

where $\sigma_2 = (id\ (dom\ \sigma_1) \oplus \{i_1 \mapsto j_1, ..., i_n \mapsto j_n\})^{\sim} \mathbin{⨟} \sigma_1$

In a schema renaming expression $e\ [j_1 \mathbin{/} i_1, ..., j_n \mathbin{/} i_n]$, expression e shall be a schema. There shall be no duplicates amongst the old names $i_1, ..., i_n$. Declarations that are merged by the renaming shall have the same type. 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.

NOTE 1 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.6.23.3 Semantics

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.

$$\begin{array}{l} [\![\ e\ [j_1 \mathbin{/} i_1, ..., j_n \mathbin{/} i_n]\ ]\!]^{\varepsilon} = \lambda\ M : Model \bullet \\ \quad \{t_1 : [\![\ e\ ]\!]^{\varepsilon} M;\ t_2 : \mathbb{W} \mid \\ \qquad t_2 = (id\ (dom\ t_1) \oplus \{i_1 \mapsto j_1, ..., i_n \mapsto j_n\})^{\sim} \mathbin{⨟} t_1 \\ \qquad \land\ t_2 \in (\_ \pfun \_) \\ \quad \bullet\ t_1\} \end{array}$$

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.6.24 Binding selection

C.6.24.1 Syntax

```
Expression      =  ...
                |  Expression , . , RefName
                |  ...
                ;
```

C.6.24.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \mathrel{\ddot{:}} [\sigma]}{\Sigma \vdash^{\varepsilon} e \,.\, i \mathrel{\ddot{:}} \sigma\ i}(i \in dom\ \sigma)$$

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.6.24.3 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.

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

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.6.25 Tuple selection

C.6.25.1 Syntax

```
Expression = ...
           | Expression , . , NUMBER
           | ...
           ;
```

C.6.25.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \mathrel{\ddot{:}} \tau_1 \times ... \times \tau_n}{\Sigma \vdash^{\varepsilon} e \,.\, b \mathrel{\ddot{:}} \tau_b}(b \in 1 \,..\, n)$$

In a tuple selection expression $e \,.\, b$, the type of expression e shall be a Cartesian product, and number b 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 tuple selection expression $e \,.\, b$ is the b'th component of the tuple that is the value of e.

$$\begin{aligned}(e \mathrel{\ddot{:}} \tau_1 \times ... \times \tau_n) \,.\, b \quad \Longrightarrow \quad & (\lambda\ i : carrier\ (\tau_1 \times ... \times \tau_n) \bullet \\ & \mu\ i_1 : carrier\ \tau_1;\ ...;\ i_n : carrier\ \tau_n \mid i = (i_1, ..., i_n) \bullet i_b)\ e\end{aligned}$$

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.6.26 Binding construction

C.6.26.1 Syntax

```
Expression = ...
           | θ , Expression , { STROKE }
           | ...
           ;
```

C.6.26.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{:} \mathbb{P}[i_1 : \tau_1; \ ...; \ i_n : \tau_n] \qquad \Sigma \vdash^{\varepsilon} i_1{}^* \mathbin{:} \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} i_n{}^* \mathbin{:} \tau_n}{\Sigma \vdash^{\varepsilon} \theta\, e\, ^* \mathbin{:} [i_1 : \tau_1; \ ...; \ i_n : \tau_n]}$$

In a binding construction expression $\theta\, e\, ^*$, the expression e shall be a schema, and in the environment shall appear names, like those in the signature of the schema but with the (optional) strokes appended, associated with the same types as those names have in the signature of schema e. The type of the whole expression is that of a binding whose signature is that of the schema.

NOTE 1 The reference expressions $i_1{}^*...i_n{}^*$ cannot refer to generic declarations, because $\tau_1...\tau_n$ are the types of schema components, which cannot be generic types.

C.6.26.3 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.

$$\theta\, e\, ^* \mathbin{:} \langle\!|\ i_1 : \tau_1; \ ...; \ i_n : \tau_n \ |\!\rangle \implies \langle\!|\ i_1 == i_1{}^*, ..., i_n == i_n{}^* \ |\!\rangle$$

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

C.6.27 Reference

C.6.27.1 Syntax

```
Expression      =  ...
                |  RefName
                |  ...
                ;
```

C.6.27.2 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.

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

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

NOTE 2 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.

$$\frac{}{\Sigma \vdash^{\varepsilon} i \mathbin{:} \tau}(i \in dom\ \Sigma)$$

where $\tau =$ if $\Sigma\ i = [i_1, ..., i_n]\tau_0$ then $\Sigma\ i$, $(\Sigma\ i)\ [\tau_1, \ ..., \ \tau_n]$ else $\Sigma\ i$

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 variable types (which the context should constrain to non-generic types). Otherwise (if the type in the environment is non-generic), that is the type of the whole expression.

NOTE 3 The operation of generic type instantiation is defined in 14.3.

NOTE 4 If the type is generic, the next phase of processing makes the implicit instantiation explicit, transforming the reference expression to a generic instantiation expression. That cannot be done here, as the new variable types $\tau_1, ..., \tau_n$ have yet to be constrained.

C.6.27.3 Semantics

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

$$i \mathrel{⦂} [i_1, ..., i_n]\tau, \tau' \implies i\ [carrier\ \tau_1, ..., carrier\ \tau_n] \mathrel{⦂} \tau'$$

$$where\ \exists_1\ \tau_1, ..., \tau_n : \mathtt{Type} \bullet \tau' = ([i_1, ..., i_n]\tau)\ [\tau_1, ..., \tau_n]$$

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 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.6.28 Generic instantiation

C.6.28.1 Syntax

```
Expression      =  ...
                |  RefName , [-tok , Expression , { ,-tok , Expression } , ]-tok
                |  ...
                ;
```

C.6.28.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathrel{⦂} \mathbb{P}\tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \mathrel{⦂} \mathbb{P}\tau_n}{\Sigma \vdash^{\varepsilon} i\ [e_1, ..., e_n] \mathrel{⦂} (\Sigma\ i)\ [\tau_1, ..., \tau_n]} (i \in dom\ \Sigma)$$

In a generic instantiation expression $i\ [e_1, ..., e_n]$, the expressions shall be sets, and the name i shall be associated with a generic type in the environment. The type of the whole expression is the instantiation of that generic type by the types of those sets.

NOTE 1 The operation of generic type instantiation is defined in 14.3.

C.6.28.3 Semantics

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.

C.6.29 Number literal

C.6.29.1 Introduction

Z accepts the ordinary notation for writing number literals that represent natural numbers, and imposes the usual meaning on those literals. The method of doing this is as follows.

The lexis defines the notion of a numeric string. The prelude defines the notions of natural number, zero, one and addition (of natural numbers). The syntactic transformation rules prescribe how numeric strings are to be understood as natural numbers, using the ideas defined in the prelude.

The extension to integers, and the introduction of other numeric operations on integers, is defined in the mathematical toolkit (annex B).

The extension to other number systems is left to user definition.

C.6.29.2 Syntax

```
Expression      =  ...
                |  NUMBER
                |  ...
                ;
```

Numeric literals are concrete expressions.

C.6.29.3 Transformation

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

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

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

C.6.30 Set extension

C.6.30.1 Syntax

```
Expression        =  ...
                  |  {-tok , [ Expression , { ,-tok , Expression } ] , }-tok
                  |  ...
                  ;
```

C.6.30.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \tau \quad \ldots \quad \Sigma \vdash^{\varepsilon} e_n \mathbin{:} \tau}{\Sigma \vdash^{\varepsilon} \{e_1, \ldots, e_n\} \mathbin{:} \mathbb{P}\tau}$$

In a set extension expression, every component expression shall be of the same type. The type of the whole expression is a set of the components' type.

C.6.30.3 Semantics

The value of the set extension expression $\{ e_1, \ldots, e_n\}$ is the set containing the values of its expressions.

$$[\![\{ e_1, \ldots, e_n\}]\!]^{\varepsilon} = \lambda M : Model \bullet \{[\![e_1]\!]^{\varepsilon} M, \ldots, [\![e_n]\!]^{\varepsilon} M\}$$

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.

C.6.31 Set comprehension

C.6.31.1 Syntax

```
Expression        =  ...
                  |  {-tok , SchemaText , • , Expression , }-tok
                  |  ...
                  ;
```

C.6.31.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\varepsilon} e_2 \mathbin{:} \tau}{\Sigma \vdash^{\varepsilon} \{e_1 \bullet e_2\} \mathbin{:} \mathbb{P}\tau}$$

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 set of the type of expression e_2, as determined in an environment overridden by the signature of schema e_1.

C.6.31.3 Semantics

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.

C.6.32 Characteristic set comprehension

C.6.32.1 Syntax

```
Expression = ...
           | ( ( {-tok , SchemaText , }-tok )   — ( {-tok , }-tok ) )
                                                — ( {-tok , Expression , }-tok )
           | ...
           ;
```

C.6.32.2 Transformation

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

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

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

C.6.33 Schema construction

C.6.33.1 Syntax

```
Expression = ...
           | ( [-tok , SchemaText , ]-tok )   — ( [-tok , Expression , ]-tok )
           | ...
           ;
```

C.6.33.2 Transformation

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.

$$[t] \quad \Longrightarrow \quad t$$

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

C.6.33.3 Type

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{:} \mathbb{P}\,\tau}{\Sigma \vdash^{\varepsilon} [i : e] \mathbin{:} \mathbb{P}[i : \tau]}$$

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 set e.

$$\frac{\Sigma \vdash^{\varepsilon} e \mathbin{:} \mathbb{P}[\sigma] \qquad \Sigma \oplus \sigma \vdash^{\mathcal{P}} p}{\Sigma \vdash^{\varepsilon} [e \mid p] \mathbin{:} \mathbb{P}[\sigma]}$$

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.

C.6.33.4 Semantics

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.

$$[\![\, [i : e] \,]\!]^{\varepsilon} \quad = \quad \lambda\ M : \mathit{Model} \bullet \{w : [\![\, e \,]\!]^{\varepsilon} M \bullet \{i \mapsto w\}\}$$

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.

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.

$$[\![[e \mid p]]\!]^{\varepsilon} \quad = \quad \lambda\ M : Model \bullet \{t : [\![e]\!]^{\varepsilon} M \mid M \oplus t \in [\![p]\!]^{\mathcal{P}} \bullet t\}$$

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.

C.6.34 Binding extension

C.6.34.1 Syntax

```
Expression = ...
           | ⟪ , [ DeclName , == , Expression , { ,-tok , DeclName , == , Expression } ] , ⟫
           | ...
           ;
```

C.6.34.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \mathbin{:} \tau_n}{\Sigma \vdash^{\varepsilon} \langle\!| i_1 == e_1, ..., i_n == e_n |\!\rangle \mathbin{:} [i_1 : \tau_1;\ ...;\ i_n : \tau_n]} (\# \{i_1, ..., i_n\} = n)$$

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

C.6.34.3 Semantics

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.

$$[\![\langle\!| i_1 == e_1, ..., i_n == e_n |\!\rangle]\!]^{\varepsilon} \quad = \quad \lambda\ M : Model \bullet \{i_1 \mapsto [\![e_1]\!]^{\varepsilon} M,\ ...,\ i_n \mapsto [\![e_n]\!]^{\varepsilon} M\}$$

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.

C.6.35 Tuple extension

C.6.35.1 Syntax

```
Expression = ...
           | (-tok , Expression , ,-tok , Expression , { ,-tok , Expression } , )-tok
           | ...
           ;
```

C.6.35.2 Type

$$\frac{\Sigma \vdash^{\varepsilon} e_1 \mathbin{:} \tau_1 \quad ... \quad \Sigma \vdash^{\varepsilon} e_n \mathbin{:} \tau_n}{\Sigma \vdash^{\varepsilon} (e_1, ..., e_n) \mathbin{:} \tau_1 \times ... \times \tau_n}$$

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.

C.6.35.3 Semantics

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.

C.6.36 Characteristic definite description

C.6.36.1 Syntax

```
Expression = ...
           | (-tok , μ , SchemaText , )-tok
           | ...
           ;
```

C.6.36.2 Transformation

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

$$(\mu\ t) \implies \mu\ t \bullet chartuple\ t$$

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

C.6.37 Parenthesized expression

C.6.37.1 Syntax

```
Expression = ...
           | (-tok , Expression , )-tok
           | ...
           ;
```

C.6.37.2 Transformation

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

$$(e) \implies e$$

It is semantically equivalent to e.

C.7 Schema text

C.7.1 Introduction

A `SchemaText` introduces local variables, with constraints on their values.

C.7.2 Syntax

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

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

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

C.7.3 Transformation

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

C.7.3.1 `Declaration`

Each declaration is transformed to an expression.

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

$$i == e \quad \Longrightarrow \quad [i : \{e\}]$$

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

$$i_1, ..., i_n : e \quad \Longrightarrow \quad [i_1 : e \mid \tau_1] \land ... \land [i_n : e \mid \tau_1]$$

C.7.3.2 `DeclPart`

Each declaration part is transformed to an expression.

$$de_1; \ ...; \ de_n \quad \Longrightarrow \quad de_1 \land ... \land de_n$$

If `NL` tokens have been used in place of any ; s, the same transformation to $\land$ applies.

C.7.3.3 `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 a schema construction expression.

$$e \mid p \quad \Longrightarrow \quad [e \mid p]$$

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

$$\Longrightarrow \quad \{\langle\!|\ |\!\rangle\}$$

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.

$$\mid p \quad \Longrightarrow \quad [\{\langle\!|\ |\!\rangle\} \mid p]$$

Annex D
(informative)

Tutorial

D.1 Introduction

The aim of this tutorial is to show, by examples, how this International Standard can be used to determine whether a specification is a well-formed Z sentence, and if it is, to determine its semantics. The examples cover some of the more difficult parts of Z, and some of the recent innovations in the Z notation.

D.2 Semantics as models

The semantics of a specification is determined by sets of models, each model being a function from names defined by the specification to values that those names are permitted to have by the constraints imposed on them in the specification. For example, consider the following definitions concerning a palette.

$$primary ::= red \mid yellow \mid blue$$

$$warmcol, coolcol : \mathbb{P}\, primary$$

$$warmcol \in \{\{red\}, \{yellow\}, \{red, yellow\}\}$$
$$coolcol \in \{\{blue\}, \{blue, yellow\}, \{blue, red\}\}$$

For the six names introduced in this specification, nine possible combinations of values are permitted by the constraints. For four of the names, the associated value is the same in all nine models.

$$\{primary \mapsto \{red, yellow, blue\}, red \mapsto red, yellow \mapsto yellow, blue \mapsto blue\}$$

The values associated with the remaining two names in the nine models are as follows.

$$\{warmcol \mapsto \{red\}, coolcol \mapsto \{blue\}\}$$
$$\{warmcol \mapsto \{red\}, coolcol \mapsto \{blue, yellow\}\}$$
$$\{warmcol \mapsto \{red\}, coolcol \mapsto \{blue, red\}\}$$
$$\{warmcol \mapsto \{yellow\}, coolcol \mapsto \{blue\}\}$$
$$\{warmcol \mapsto \{yellow\}, coolcol \mapsto \{blue, yellow\}\}$$
$$\{warmcol \mapsto \{yellow\}, coolcol \mapsto \{blue, red\}\}$$
$$\{warmcol \mapsto \{red, yellow\}, coolcol \mapsto \{blue\}\}$$
$$\{warmcol \mapsto \{red, yellow\}, coolcol \mapsto \{blue, yellow\}\}$$
$$\{warmcol \mapsto \{red, yellow\}, coolcol \mapsto \{blue, red\}\}$$

All nine models of this specification also associate values with the names defined in the prelude (clause 11), since the prelude is implicitly present in every specification.

This International Standard specifies the relation between Z specifications and their semantics in terms of sets of models. That relation is specified as a composition of relations, each implementing a phase within the standard. Those phases are as identified in Figure 1 in the conformance clause, namely mark-up, lexing, parsing, characterising, syntactic transformation, type inference, instantiating, semantic transformation and semantic relation. The rest of this tutorial illustrates the effects of those phases on example Z phrases.

D.3 Given types and schema definition paragraphs

The following two paragraphs are taken from the birthday book specification [15].

$$[NAME, DATE]$$

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

$$known = dom\ birthday$$

The mark-up, lexing, parsing and syntactic transformation phases are illustrated using this example.

D.3.1 Mark-ups

The mathematical representation of Z is what one would write with pen, pencil, chalk, etc. Instructing a computer to produce the same appearance currently requires the use of a mark-up language. There are many different mark-up languages, each tailored to different circumstances, such as particular typesetting software. This International Standard defines some mark-ups in annex A, by relating substrings of the mark-up language to strings of Z characters. Source text for the birthday-book paragraphs written in the mark-ups defined in annex A follow. The translation of these into sequences of Z characters is not explained here – annex A provides sufficient information.

D.3.1.1 LaTeX mark-up

```
\begin{zed}
[NAME, DATE]
\end{zed}

\begin{schema}{BirthdayBook}
known : \power NAME\\
birthday : NAME \pfun DATE
\where
known = \dom~birthday
\end{schema}
```

D.3.1.2 Email mark-up

```
[NAME, DATE]

+-- BirthdayBook ---
known : %P NAME
birthday : NAME -+-> DATE
|--
known = dom birthday
---
```

D.3.2 Lexing

Lexing is the translation of a specification's sequence of Z characters to a corresponding sequence of tokens. The translation is defined by the lexis in clause 7. Associated with the tokens `NAME` and `NUMBER` are the original names and numbers. Here on the left is the sequence of tokens corresponding to the extract from the birthday book (with `-tok` suffices omitted), and on the right is the same sequence but revealing the underlying spelling of the name tokens.

[NAME, NAME] END	[*NAME*, *DATE*] END
SCH NAME	SCH *BirthdayBook*
NAME : PRE NAME NL	*known* : $\mathbb{P}$ *NAME* NL
NAME : NAME I NAME	*birthday* : *NAME* $\pfun$ *DATE*
|	|
NAME = NAME NAME	*known* = *dom birthday*
END	END

The layout here is of no significance: there are `NL` and `END` tokens where ones are needed. The paragraph outline has been replaced by a `SCH` box token, to satisfy the linear syntax requirement of the syntactic metalanguage. `NAME` and `I` are name tokens: this abstraction allows the fixed size grammar of the concrete syntax to cope with the extensible Z notation.

This specification's sequence of Z characters does conform to the lexis. If it had not, then subsequent phases would not be applicable, and this International Standard would not define a meaning for the specification.

D.3.3 Parsing

Parsing is the translation of the sequence of tokens produced by lexing to a tree structure, grouping the tokens into grammatical phrases. The grammar is defined by the concrete syntax in clause 8. The parse tree for the birthday-book specification is shown in Figure D.1.

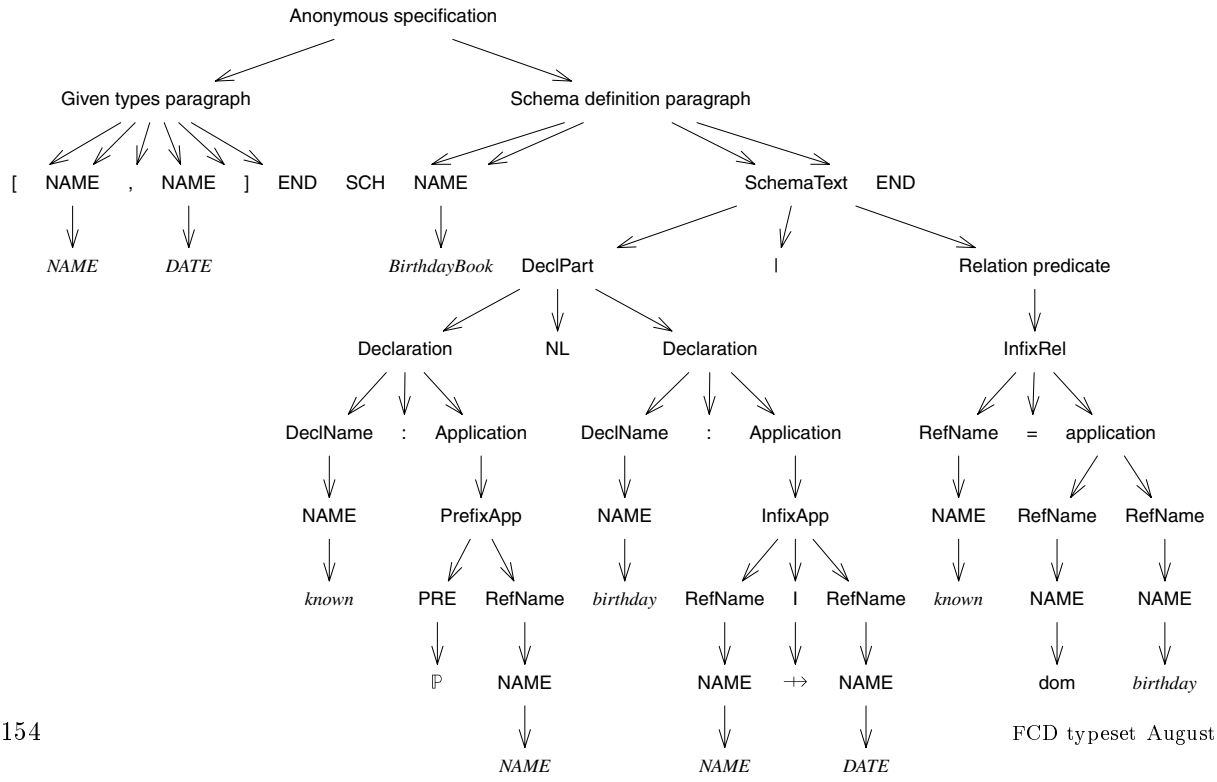
The sequence of tokens produced by lexing can be seen in this tree by reading just the leaf nodes in order from left to right. To save space elsewhere in this International Standard, parse trees are presented as just their textual fringes, with parentheses added where necessary to override precedences.

This specification's sequence of tokens does conform to the concrete syntax. If it had not, then subsequent phases would not be applicable, and this International Standard would not define a meaning for the specification.

D.3.4 Syntactic transformation

The meaning of a Z specification is established by relating it to an interpretation in a semantic universe. That relation is expressed using ZF set theory, which is not itself formally defined. It is therefore beneficial to define as much Z notation as possible by transformations to other Z notation, so that only a relatively small kernel need be related using ZF set theory. Conveniently, that Z kernel contains largely notation that has direct counterparts

Figure D.1 – Parse tree of birthday book example



in traditional ZF set theory, the novel Z notation having been largely transformed away. A further benefit is that the transformations reveal relationships between different Z notations. The syntactic transformation stage is one of several phases of such transformation.

The syntactic transformation rules (clause 12) are applied to a parsed sentence of the concrete syntax (clause 8). The notation that results is a sentence of the annotated syntax (clause 10).

Consider the effect of the syntactic transformation rules on the birthday book extract. There is no syntactic transformation rule for given types; given types are in the annotated syntax. So the first paragraph is left unchanged.

[NAME,NAME] END	$[NAME, DATE]$ END

The schema paragraph requires several syntactic transformations before it becomes a sentence of the annotated syntax. The order in which these syntactic transformations are applied does not matter, as the same result is obtained.

Transform NL by first rule in 12.2.6.

SCH NAME	SCH *BirthdayBook*
NAME : PRE NAME;	$known : \mathbb{P}\, NAME$;
NAME : NAME I NAME	$birthday : NAME \pfun DATE$
|	|
NAME = NAME NAME	$known = dom\ birthday$
END	END

Transform application $\mathbb{P}\, NAME$ by sixth PrefixApp rule in 12.2.11.

SCH NAME	SCH *BirthdayBook*
NAME : PRE NAME;	$known : \mathbb{P}\, NAME$;
NAME : NAME I NAME	$birthday : NAME \pfun DATE$
|	|
NAME = NAME NAME	$known = dom\ birthday$
END	END

Transform generic application $NAME \pfun DATE$ by sixth InfixApp rule in 12.2.11.

SCH NAME	SCH *BirthdayBook*
NAME : $\mathbb{P}$ NAME;	$known : \mathbb{P}\, NAME$;
NAME : NAME [NAME, NAME]	$birthday : \bowtie \pfun \bowtie [NAME, DATE]$
|	|
NAME = NAME NAME	$known = dom\ birthday$
END	END

Transform equality by third InfixRel rule in 12.2.10.

SCH NAME	SCH *BirthdayBook*
NAME : $\mathbb{P}$ NAME;	$known : \mathbb{P}\, NAME$;
NAME : NAME [NAME, NAME]	$birthday : \bowtie \pfun \bowtie [NAME, DATE]$
|	|
NAME $\in$ {NAME NAME}	$known \in \{dom\ birthday\}$
END	END

Transform basicdecls by sixth rule in 12.2.6.

```
SCH NAME                                  SCH BirthdayBook
[NAME : ℙ NAME];                          [known : ℙ NAME];
[NAME : NAME [NAME, NAME]]                [birthday : ⋈ ⇸ ⋈ [NAME, DATE]]
|                                         |
NAME ∈ {NAME NAME}                        known ∈ {dom birthday}
END                                       END
```

Transform schema text by second rule in 12.2.6.

```
SCH NAME                                  SCH BirthdayBook
[[NAME : ℙ NAME] ∧                        [[known : ℙ NAME] ∧
[NAME : NAME [NAME, NAME]]                [birthday : ⋈ ⇸ ⋈ [NAME, DATE]]
|                                         |
NAME ∈ {NAME NAME}]                       known ∈ {dom birthday}]
END                                       END
```

Transform paragraph by first rule in 12.2.3.

```
AX                                        AX
[NAME ==                                  [BirthdayBook ==
[[NAME : ℙ NAME] ∧                        [[known : ℙ NAME] ∧
[NAME : NAME [NAME, NAME]]                [birthday : ⋈ ⇸ ⋈ [NAME, DATE]]
|                                         |
NAME ∈ {NAME NAME}]]                      known ∈ {dom birthday}]
END                                       END
```

The two paragraphs now form a sentence of the annotated syntax. These syntactic transformations do not change the meaning: the meaning of the annotated representation is the same as that of the original schema paragraph. This is ensured, despite transformations to notations of different precedences, by transforming trees not text – the trees are presented as text above solely to save space.

Do not be surprised that the result of syntactic transformation looks "more complicated" than the original formulation – if it did not, there would not have been much point in having the notation that has been transformed away. The benefits are that fewer notations remain to be defined, and those that have been defined have been defined entirely within Z.

D.4 Axiomatic description paragraphs

Here is a very simple axiomatic description paragraph, preceded by an auxiliary given types paragraph.

$$[X]$$

$$i : X$$

D.4.1 Lexing and parsing

Lexing generates the following sequence of tokens, with corresponding spellings of name tokens.

```
[NAME] END                                [X] END
AX NAME : NAME END                        AX i : X END
```

Parsing proceeds as for the birthday book, so is not explained in detail again.

D.4.2 Syntactic transformation

The schema text is transformed to an expression.

```
AX [NAME : NAME] END                      AX [i : X] END
```

This is now a sentence of the annotated syntax.

D.4.3 Type inference

The type inference phase adds annotations to each expression and each paragraph in the parse tree. Without going into too much detail, the signature of the given types paragraph is determined by rule 13.2.3.1 to be $[X : \mathbb{P}(\texttt{GIVEN}\ X)]$.

$([X] \ \fatsemi\ [X : \mathbb{P}(\texttt{GIVEN}\ X)])\ \texttt{END}$

Rule 13.2.2.1 adds the name of the given type X to the type environment, associated with type $\mathbb{P}(\texttt{GIVEN}\ X)$. The annotation for the reference expression referring to X is determined by rule 13.2.5.1 using that type environment to be $\mathbb{P}(\texttt{GIVEN}\ X)$. Hence the type of the variable construction expression is found by rule 13.2.5.13 to be $\mathbb{P}[i : \texttt{GIVEN}\ X]$. Hence the signature of the axiomatic description paragraph is determined. The resulting annotated tree is shown in Figure D.2, and as linear text as follows.

$(\texttt{AX}\ ([i : (X \ \fatsemi\ \mathbb{P}(\texttt{GIVEN}\ X))] \ \fatsemi\ \mathbb{P}[i : \texttt{GIVEN}\ X]) \ \fatsemi\ [i : \texttt{GIVEN}\ X])\ \texttt{END}$

This specification's parse tree is well-typed. If it were not, then subsequent phases would not be applicable, and this International Standard would not define a meaning for the specification.

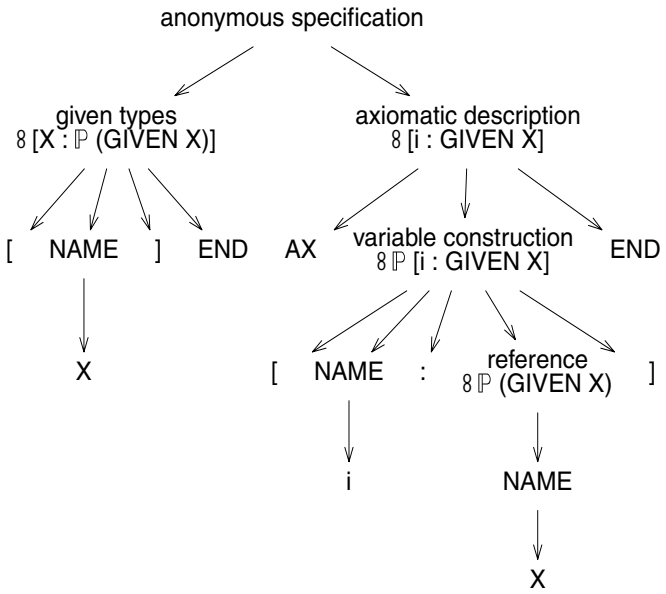
D.4.4 Semantic relation

The semantic relation phase takes a sentence of the annotated syntax and relates it to its meaning in terms of sets of models. The meaning of a paragraph d is given by the semantic relation $[\![\ d\ ]\!]^{\mathcal{D}}$, which relates a model to that same model extended with associations between its names and their semantic values in the given model. For the example given types paragraph, the semantic relation in 16.2.3.1 relates any model to that model extended with an association of the given type name X with an arbitrarily chosen set w. (A further association is made between a distinctly decorated version of the given type name $X\heartsuit$ and that same semantic value, for use in avoiding variable capture.)

$\{X \mapsto w, X\heartsuit \mapsto w\}$

This is one model of the prefix of the specification up to the given types paragraph (ignoring the prelude). The set of models defining the meaning of this prefix includes other models, each with a different set w.

Figure D.2 – Annotated parse tree of part of axiomatic example



The meaning of an expression e is given by the semantic function $[\![e]\!]^{\varepsilon}$, which maps the expression to its semantic value in a given model. Within the axiomatic description paragraph, the reference to the given type X has a semantic value determined by relation 16.2.5.1 as being the semantic value w already associated with the given type name X in the model. The variable construction expression $[i : X]$ has a semantic value determined by relation 16.2.5.9 that represents a set of bindings of the name i to members of the semantic value of the reference to X, i.e. the set w. The meaning of the example axiomatic description paragraph, as given by semantic relation 16.2.3.2, is to relate any model to that model extended with a binding that is in the set that is the semantic value of the variable construction expression. So, if the members of w are $w_1, w_2, ...$, then the set of models defining the meaning of the specification includes the following.

$\{\{X \mapsto w, X\heartsuit \mapsto w, i \mapsto w_1\}, \{X \mapsto w, X\heartsuit \mapsto w, i \mapsto w_2\}, ...\}$

D.5 Generic axiomatic description paragraphs

Here is a generic axiomatic description paragraph. Although it looks simple, it has the complication of being a loose generic definition.

$$\begin{array}{|l} [X] \\ \hline i : X \end{array}$$

D.5.1 Lexing and parsing

Lexing generates the following sequence of tokens, with corresponding spellings of name tokens.

`GENAX [NAME] NAME : NAME END`	`GENAX [X] i : X END`

Parsing proceeds as for the birthday book, so is not explained in detail again.

D.5.2 Syntactic transformation

The schema text is transformed to an expression.

`GENAX [NAME] [NAME : NAME] END`	`GENAX [X] [i : X] END`

This is now a sentence of the annotated syntax.

D.5.3 Type inference

Without going into too much detail, rule 13.2.3.3 adds the name of the generic parameter X to the type environment, associated with type $\mathbb{P}$(`GENTYPE` X). The annotation for the reference expression referring to X is determined by rule 13.2.5.1 using that type environment to be $\mathbb{P}$(`GENTYPE` X). Hence the type of the variable construction expression is found by rule 13.2.5.13 to be $\mathbb{P}[i :$ `GENTYPE` $X]$. Hence the signature of the generic axiomatic description paragraph is determined.

(`GENAX` $[X]$ $([i : (X \ \fatsemi\ \mathbb{P}$(`GENTYPE` $X))] \ \fatsemi\ \mathbb{P}[i :$ `GENTYPE` $X]) \ \fatsemi\ [i :$ `GENTYPE` $X])$ `END`

D.5.4 Semantic relation

Every use of a generic definition instantiates the generic parameters with particular sets. A suitable semantic value for a generic definition is therefore a function from the semantic values of the sets instantiating the generic parameters to the semantic value of the definition given those values for the parameters. In the case of a loose generic definition, several models are needed to express its semantics, each giving a different function. Each model defining the meaning of the example generic axiomatic description paragraph has the following form.

$\{i \mapsto \{s_1 \mapsto \mathit{member\ of}\ s_1,\ s_2 \mapsto \mathit{member\ of}\ s_2,\ ...\}\}$

The model associates the name i with a function from sets $s_1, s_2, ...$ instantiating the generic parameter X to the semantic value of i resulting from the corresponding instantiation. Different models for this paragraph have different members of $s_1, s_2, ...$ The way this is defined by the semantic relations is roughly as follows.

The semantic relation in 16.2.3.3 specifies that w is a binding in the semantic value of schema e in the given model extended with an association of the generic parameter X with the set instantiating it w_1. (The model is

also extended with a further association between a distinctly decorated version of the generic parameter $X\spadesuit$ and that same semantic value w_1, for use in avoiding variable capture). This is specified in a way that is cautious of e being undefined in the extended model. The determination of the value of e is done in the same way as for the preceding example, using semantic relations 16.2.5.1 and 16.2.5.9. The semantic relation for the paragraph is then able to extend its given model with the association illustrated above.

D.6 Operator templates and generics

The definition of relations in the toolkit provides an example of an operator template and the definition of a generic operator.

generic 5 rightassoc $(\_ \leftrightarrow \_)$

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$

D.6.1 Lexing and parsing

An operator template paragraph affects the lexing and parsing of subsequent paragraphs. In this example, it causes subsequent appearances of names using the word $\leftrightarrow$ to be lexed as `I` tokens, and hence its infix applications are parsed as operator names (illustrated in the example) or as generic operator application expressions. An operator template paragraph does not have any further effect on the meaning of a specification, so a parsed representation is needed of only the generic operator definition paragraph, for which lexing generates the following sequence of tokens and corresponding spellings of name tokens.

`NAME I NAME` $==$ `PRE(NAME` $\times$ `NAME) END` $\qquad$ $X \leftrightarrow Y == \mathbb{P}(X \times Y)$ `END`

Parsing proceeds as for the birthday-book example, so is not explained in detail again.

D.6.2 Syntactic transformation

The generic operator name is transformed by the first `InfixGenName` rule in 12.2.9.

$\_$ `I` $\_$ `[NAME, NAME]` $==$ $\mathbb{P}$`(NAME` $\times$ `NAME) END` $\qquad$ $\_ \leftrightarrow \_ [X, Y] == \mathbb{P}(X \times Y)$ `END`

This generic horizontal definition paragraph is then transformed by 12.2.3.4 to a generic axiomatic description paragraph, which is the sole form of generic definition for which there is a semantic relation.

```
GENAX [NAME, NAME]
_ I _ == ℙ(NAME × NAME)
END
```

```
GENAX [X, Y]
_ ↔ _ == ℙ(X × Y)
END
```

Transform Cartesian product expression by the rule in 12.2.5.8 to a set of pairs.

```
GENAX [NAME, NAME]
_ I _ == ℙ{NAME : NAME; NAME : NAME • (NAME, NAME)}
END
```

```
GENAX [X, Y]
_ ↔ _ == ℙ{x : X; y : Y • (x, y)}
END
```

Transform the operator name by the first `InfixName` rule in 12.2.8.3.

```
GENAX [NAME, NAME]
NAME == ℙ{NAME : NAME; NAME : NAME • (NAME, NAME)}
END
```

```
GENAX [X, Y]
⋈↔⋈ == ℙ{x : X; y : Y • (x, y)}
END
```

Transform the schema texts to expressions.

```
GENAX [NAME, NAME]
[NAME : {ℙ{[NAME : NAME] ∧ [NAME : NAME] • (NAME, NAME)}}]
END
```

```
GENAX [X, Y]
[⋈↔⋈ : {ℙ{[x : X] ∧ [y : Y] • (x, y)}}]
END
```

This is now a sentence of the annotated syntax.

D.6.3 Type inference

The type inference phase adds annotations to the parse tree. For this example, the resulting subtree for the set extension expression (to the right of the colon) is shown in Figure D.3.

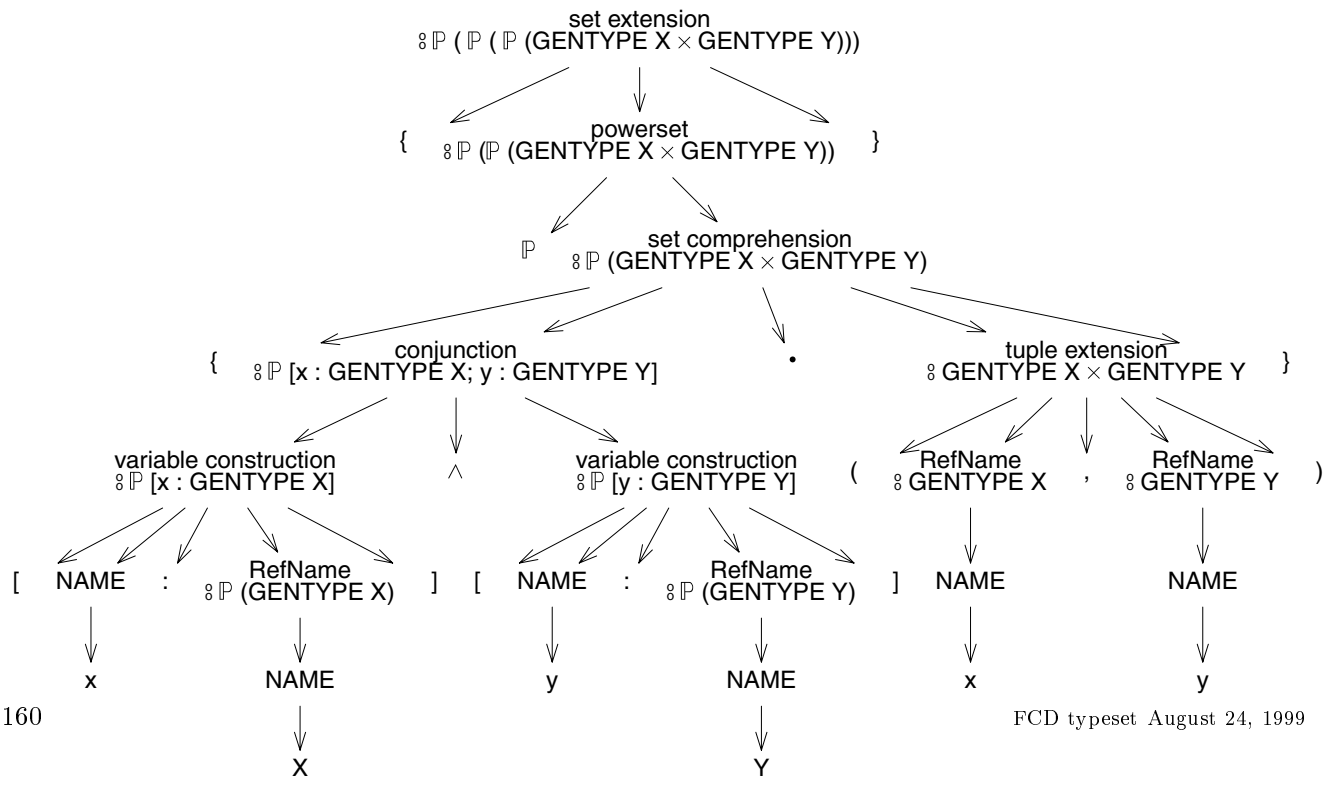
Informally, the way the annotations are determined is as follows. The type inference rule for generic axiomatic description paragraph (13.2.3.3) overrides the type environment with types for the generic parameters X and Y. The type inference rule for reference expression (13.2.5.1) retrieves these types for the references to X and Y. The type inference rule for variable construction expression (13.2.5.13) is then able to build the schema types. The type inference rule for schema conjunction expression (13.2.5.16) merges those schema types. The type inference rule for set comprehension expression (13.2.5.4) overrides the type environment with types for the local declarations x and y. The type inference rule for reference expression (13.2.5.1) retrieves these types for the references to x and y. The type inference rule for tuple extension expression (13.2.5.6) is then able to build the Cartesian product type. The type of the set comprehension is thus determined, and hence that of the powerset by rule 13.2.5.5 and that of the set extension by rule 13.2.5.3.

D.6.4 Semantic relation

For the example generic axiomatic description paragraph, the semantic relation in 16.2.3.3 associates name $\_ \leftrightarrow \_$ with a function from the semantic values of the sets instantiating the generic parameters X and Y to the semantic value of the powerset expression given those values for X and Y. The semantic value of the example's powerset expression is given by semantic relations 16.2.5.4, 16.2.5.5 and 16.2.5.6 as sets of tuples in ZF set theory. Hence the example generic axiomatic description paragraph adds the following association to the meaning of the specification.

$$(\_ \leftrightarrow \_) \mapsto \{(\textit{set for } X, \textit{set for } Y) \mapsto \textit{value of powerset expression given that } X \textit{ and } Y, \textit{ and so on for all combinations of sets for } X \textit{ and } Y\}$$

Figure D.3 – Annotated parse tree of part of generic example



Syntactic transformation 12.2.3.4 moved the name of the generic operator to after the generic parameters. In constructing this association, the name had to be lifted back out again. This has sometimes been called the generic lifting operation.

D.7 Mutually recursive free types

The standard notation for free types is an extension of the traditional notation, to allow the specification of mutually recursive free types, such as the following example.

$$
\begin{array}{l}
exp ::= Node\langle\!\langle \mathbb{N}_1 \rangle\!\rangle \\
\qquad \mid Cond\langle\!\langle pred \times exp \times exp \rangle\!\rangle \\
\& \\
pred ::= Compare\langle\!\langle exp \times exp \rangle\!\rangle
\end{array}
$$

This specifies a tiny language, in which an expression *exp* can be a conditional involving a predicate *pred*, and a *pred* compares expressions. A more realistic example would have more kinds of expressions and predicates, and maybe other auxiliary types perhaps in mutual recursion with these two, but this small example suffices here.

Like the previous examples, the source text for this one has to be taken through the phases of mark-up, lexing, parsing, syntactic transformation and type inference. (There are no applicable characterisations or instantiations.) The focus here is on the semantic transformation of free types. (Strictly, the Cartesian products should be syntactically transformed first, but keeping them makes the following more concise.)

D.7.1 Semantic transformation

Transforming the above free type by rule 12.2.3.5 generates the following Z notation. The semantic transformation rules are defined in terms of concrete notation for clarity, which should itself be subjected to further transformations, though that is not done here.

D.7.1.1 Type declarations

$$[exp, pred]$$

D.7.1.2 Membership constraints

$$
\begin{array}{l}
Node : \mathbb{P}(\mathbb{N}_1 \times exp) \\
Cond : \mathbb{P}((pred \times exp \times exp) \times exp) \\
Compare : \mathbb{P}((exp \times exp) \times exp)
\end{array}
$$

D.7.1.3 Total functionality constraints

$$
\begin{array}{l}
\forall\, u : \mathbb{N}_1 \bullet \exists_1\, x : Node \bullet x.1 = u \\
\forall\, u : pred \times exp \times exp \bullet \exists_1\, x : Cond \bullet x.1 = u \\
\forall\, u : exp \times exp \bullet \exists_1\, x : Compare \bullet x.1 = u
\end{array}
$$

D.7.1.4 Injectivity constraints

$$
\begin{array}{l}
\forall\, u, v : nat_1 \mid Node\ u = Node\ v \bullet u = v \\
\forall\, u, v : pred \times exp \times exp \mid Cond\ u = Cond\ v \bullet u = v \\
\forall\, u, v : exp \times exp \mid Compare\ u = Compare\ v \bullet u = v
\end{array}
$$

D.7.1.5 Portmanteau disjointness constraint

There are no disjointness constraints from the *pred* type as it has only one injection and no element values.

$$
\begin{array}{l}
\forall\, b_1, b_2 : \mathbb{N} \bullet \\
\quad \forall\, w : exp \mid \\
\qquad (b_1 = 1 \land w \in \{x : Node \bullet x.2\} \lor \\
\qquad\quad b_1 = 2 \land w \in \{x : Cond \bullet x.2\}) \\
\qquad \land\ (B_2 = 1 \land w \in \{x : Node \bullet x.2\} \lor \\
\qquad\quad b_2 = 2 \land w \in \{x : Cond \bullet x.2\}) \bullet \\
\qquad\quad b_1 = b_2
\end{array}
$$

D.7.1.6 Induction constraint

$$
\begin{array}{l}
\forall\, w_1 : \mathbb{P}\, exp;\ w_2 : \mathbb{P}\, pred \mid \\
\qquad (\forall\, y : (\mu\, exp == w_1;\ pred == w_2 \bullet \mathbb{N}_1) \bullet \\
\qquad\quad Node\ y \in w_1) \land \\
\qquad (\forall\, y : (\mu\, exp == w_1;\ pred == w_2 \bullet pred \times exp \times exp) \bullet \\
\qquad\quad Cond\ y \in w_1) \land \\
\qquad (\forall\, y : (\mu\, exp == w_1;\ pred == w_2 \bullet exp \times exp) \bullet \\
\qquad\quad Compare\ y \in w_2) \bullet \\
\quad w_1 = exp \land w_2 = pred
\end{array}
$$

D.8 Chained relations and implicit generic instantiation

The semantics of chained relations is defined to give a meaning to this example,

$$primary \neq \varnothing \subseteq warmcol$$

in which $\varnothing$ refers to the generic definition of empty set and so is implicitly instantiated, whilst rejecting the following example as being not well-typed,

$$primary \neq \varnothing \subseteq \mathbb{A}$$

because the single $\varnothing$ expression in the second example needs to be instantiated differently for the two relations.

To demonstrate how this is done, the former example is taken through syntactic transformation, type inference, and instantiation.

D.8.1 Syntactic transformation

The chaining is transformed by the first `InfixRel` rule in 12.2.10 to a conjunction of relations in which the duplicates of the common expression are constrained to be of the same type by giving them the same annotation.

$$primary \neq (\varnothing \mathbin{:} \tau) \land (\varnothing \mathbin{:} \tau) \subseteq warmcol$$

The third `InfixRel` rule in 12.2.10 transforms these two relations to membership predicates.

$$(primary, (\varnothing \mathbin{:} \tau)) \in (\_ \neq \_) \land ((\varnothing \mathbin{:} \tau), warmcol) \in (\_ \subseteq \_)$$

The two operator names are transformed by the second rule in 12.2.8.3.

$$(primary, (\varnothing \mathbin{:} \tau)) \in \bowtie\!\neq\!\bowtie \land ((\varnothing \mathbin{:} \tau), warmcol) \in \bowtie\!\subseteq\!\bowtie$$

This is now a phrase of the annotated syntax.

D.8.2 Type inference

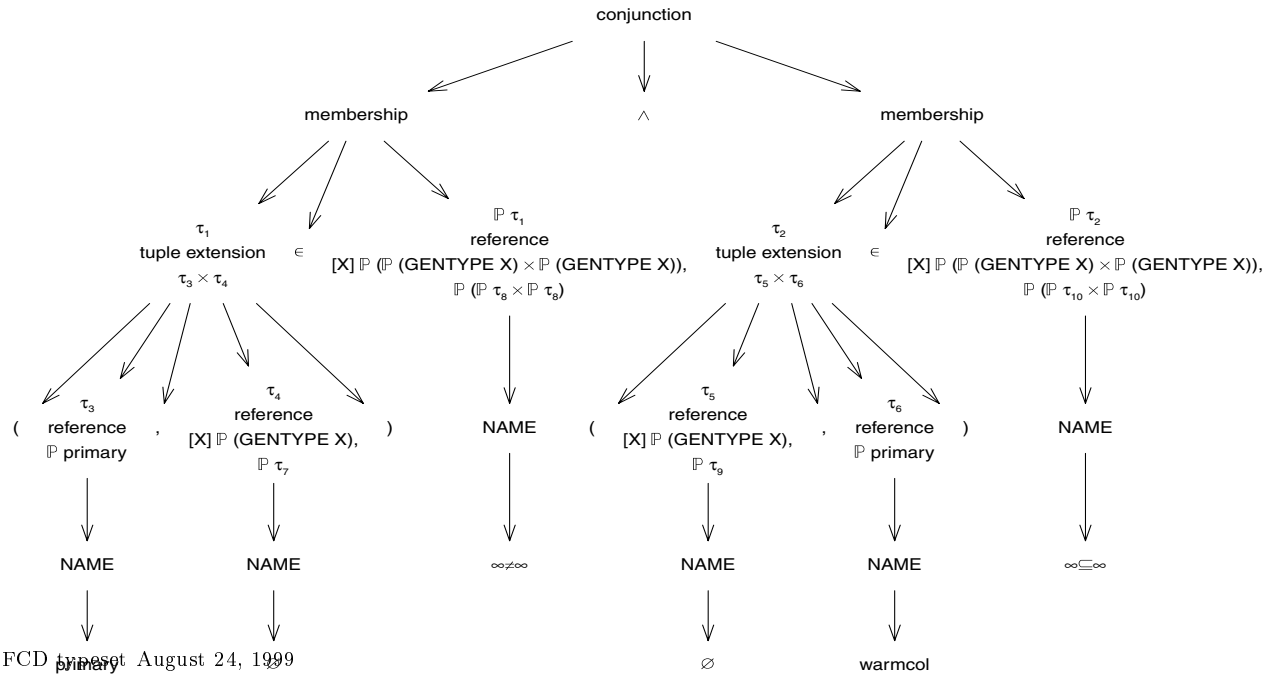
The effect of the type inference rules on this example phrase is illustrated in Figure D.4. (The tool used to draw that figure has no $\bowtie$ symbol, so ∞ is used instead.)

In this tree, each expression node has two types: the one above the node is the type that the expression is expected to have given the context in which it appears, i.e. imposed by the type inference rule for the phrase of which the expression is a part, and the one below the node is the type inferred for this expression, i.e. by the type inference rule for that expression. In detail, the first membership's type inference rule (13.2.4.1) contrains its expressions to be of types τ_1 and $\mathbb{P}\,\tau_1$. The first tuple extension's type inference rule (13.2.5.6) constrains its expected type τ_1 to be a Cartesian product type $\tau_3 \times \tau_4$, where τ_3 and τ_4 are the expected types of the two reference expressions in the tuple. The first reference expression's type inference rule (13.2.5.1) constrains its expected type τ_3 to the type associated with the referenced name (*primary*) in the type environment. The second reference expression's type inference rule (13.2.5.1) behaves slightly differently, as the type associated with the referenced name ($\varnothing$) is a generic type. That generic type $[X]\,\mathbb{P}\,X$ is noted on the node for use by the following Instantiation phase, and the type inferred for this reference expression is an instantiation of this generic type with new distinct variable types, i.e. $\mathbb{P}\,\tau_7$. The reference to the generic inequality declaration is treated similarly. These type constraints are sufficient to determine a unique assignment of type annotations to all of the expressions in the first membership. The second membership is typechecked similarly. The constraint imposed by the chained relation between the types of the two references to $\varnothing$ (not shown in the figure) is satisfied.

D.8.3 Instantiation

Those reference expressions that refer to generic definitions have to be transformed to generic instantiation expressions for their meaning to be determined. This is done by the instantiation rule (14.4). It determines the generic instantiations by comparison of the generic type with the inferred type. For example, the references to $\varnothing$ have been given the type annotation $\mathbb{P}$(`GIVEN` *primary*), which is the instance of $[X]\,\mathbb{P}\,X$ in which X is `GIVEN` *primary*. Hence the instantiation rule effects the following transformation.

Figure D.4 – Type constraints for chained relation example



$$\varnothing \mathrel{:} [X]\, \mathbb{P}\, X, \mathbb{P}(\texttt{GIVEN}\ primary) \quad \Longrightarrow \quad \varnothing[primary \mathrel{:} \mathbb{P}(\texttt{GIVEN}\ primary)] \mathrel{:} \mathbb{P}(\texttt{GIVEN}\ primary)$$

D.9 Logical inference rules

This document does not attempt to standardise any particular deductive system for Z. However, the soundness of potential logical inference rules can be shown relative to the sets of models defined by the semantics. Some examples are given here.

The predicate true can be used as an axiom. The proof of this is trivial: an axiom p is sound if and only if $[\![\ p\ ]\!]^{\mathcal{P}} = Model$ (as given by the definition of soundness in 5.2.3), and from the semantic relation for truth predicates (16.2.4.2), $[\![\ \text{true}\ ]\!]^{\mathcal{P}} = Model$.

The inference rule with premiss $\neg\ \neg\ p$ and consequent p is sound if and only if

$$[\![\ \neg\ \neg\ p\ ]\!]^{\mathcal{P}} \subseteq [\![\ p\ ]\!]^{\mathcal{P}}$$

(again as given by the definition of soundness in 5.2.3). By two applications of the semantic relation for negation predicate (16.2.4.3), this becomes

$$Model \setminus (Model \setminus [\![\ p\ ]\!]^{\mathcal{P}}) \subseteq [\![\ p\ ]\!]^{\mathcal{P}}$$

which by properties of set difference becomes

$$[\![\ p\ ]\!]^{\mathcal{P}} \subseteq [\![\ p\ ]\!]^{\mathcal{P}}$$

which is a property of set inclusion.

The transformation rules of clauses 12 and 15 inspire corresponding logical inference rules: any logical inference rule whose sole premiss matches the left-hand side of a transformation rule and whose consequent is the corresponding instantiation of that transformation rule's right-hand side is sound.

Annex E
(informative)

Conventions for state-based descriptions

E.1 Introduction

This annex records some of the conventions of notation that are often used when state-based descriptions of systems are written in Z. Conventions for identifying before and after states (x and x'), operations on those states (ΔS and ΞS) and input and output variables ($i?$ and $o!$) are given.

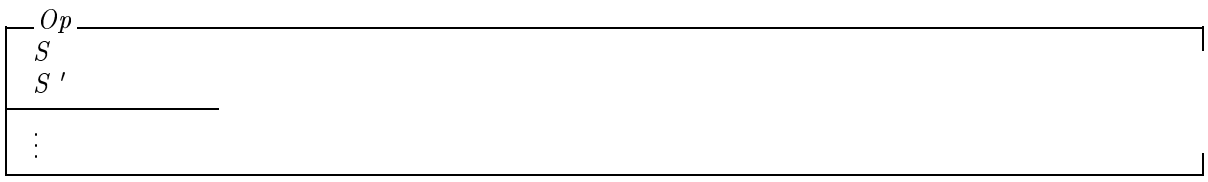
E.2 States

When giving a model-based description of a system, the state of the system and the operations on the state are specified. Each operation is described as a relation between states. It is therefore necessary to distinguish between the values of state variables before the operation and their values afterwards. The convention in Z is to use dashes (primes) to make this distinction: if the state variables are x and y, then a predicate describing an operation is written using the variables x, y, x', y', where x and y denote the values before the operation, and x' and y' denote the values afterwards. (The predicate can also refer to any global constants.) For instance, if x and y are both integer variables, then an operation which incremented both variables could be specified as follows.

$$x' = x + 1 \land y' = y + 1$$

In order to use predicates like this to describe operations, all of the variables have to be in scope. If the state has been described in a schema S, then including S in the declaration part of the operation schema brings the state variables — x and y in the example above — into scope. The after-state variables are similarly introduced by including S': this is a schema obtained from S by adding a dash to all the variables in the signature of S, and replacing every occurrence of such a variable in the predicate part of S by its dashed counterpart. Notice that the variables from the signature of S are the only ones which are dashed — global constants, types etc remain undashed. If S contains a variable which has already been decorated in some way, then an extra dash is added to the existing decoration.

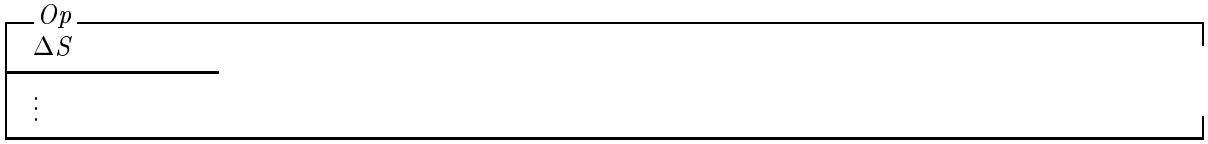
Thus operations can be described in Z by a schema of the form



Since the inclusion of undashed and dashed copies of the state schema is so common, an abbreviation is used:

$$\Delta S == [S;\ S']$$

The operation schema above now becomes



It should be stressed that this use of Δ is only a convention — Δ is not an 'operator on schemas', merely a character in the schema name. One reason for this is that some authors like to include additional invariants in

their Δ-schemas. For instance, if S contained an additional component z, but none of the operations ever changed z, then ΔS could be defined by

$$\Delta S == [S;\ S' \mid z' = z] \quad ,$$

thus making it unnecessary to include $z' = z$ in each operation description. If a name ΔS is referred to without a declaration of it having appeared previously, the reference is treated as being equivalent to $[S;\ S']$.

It should be noted that strange results can occur if this conventional definition of ΔS is used on a schema S that contains variables which are not intended to be state components, perhaps inputs or outputs (see below). The sequence of decorations after a variable name might then become difficult to interpret.

There is one further piece of notation for describing state transitions: when enquiry operations are being described, it is often necessary to specify that the state should not change. For this the Ξ-convention is used. Unless it has been explicitly defined to mean something else, references to ΞS are treated as being equivalent to $[S;\ S' \mid \theta S = \theta S']$. Note that ΞS is not defined in terms of ΔS, in case ΔS has been given an explicit unconventional definition.

E.3 Inputs and outputs

For many systems, it is convenient to be able to describe operations not just in terms of relations between states, but with inputs and outputs as well. The input values of an operation are provided by 'the environment', and the outputs are returned to the environment.

In order to distinguish a variable intended as either an input or an output in an operation schema from a state-before variable (which has no decoration), an additional suffix is used: ? for input variables and ! for output variables. Thus *name*? denotes an input, and *result*! denotes an output.

Bibliography

[1] Goldfarb, C.F., *The SGML handbook* Clarendon Press, Oxford, 1990

[2] Enderton, H.B., *Elements of Set Theory* Academic Press, 1977

[3] Hayes, I. (editor) *Specification Case Studies* Prentice-Hall, first edition, 1987

[4] Hayes, I. (editor) *Specification Case Studies* Prentice-Hall, second edition, 1993

[5] ISO/IEC 646:1991 *Information technology – ISO 7-bit coded character set for information interchange* 3rd edition

[6] ISO 8879:1986(E) *Information Processing — Text and Office Systems — Standard Generalized Mark-up Language (SGML)*

[7] ISO/IEC 10646-1:1993 *Information Technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

[8] ISO/IEC 14977:1996 *Information Technology — Syntactic Metalanguage — Extended BNF*

[9] King, S., Sørensen, I.H. and Woodcock, J.C.P. *Z: Grammar and Concrete and Abstract Syntaxes* PRG-68, Programming Research Group, Oxford University, July 1988

[10] Lalonde, W.R. and des Rivieres, J. *Handling Operator Precedence in Arithmetic Expressions with Tree Transformations* ACM *Transactions on Programming Languages and Systems*, 3(1) January 1981

[11] Lamport, L *LaTeX: A Document Preparation System — User's Guide and Reference Manual* Addison-Wesley, second edition, 1994

[12] Sperberg-McQueen, C.M. and Burnard, Lou (editors), *Guidelines for Electronic Text Encoding and Interchange, TEI P3,* Text Encoding Initiative, 1994

[13] Spivey, J.M., *Understanding Z* Cambridge University Press, 1988

[14] Spivey, J.M., *The Z Notation — A Reference Manual* Prentice-Hall, first edition, 1989

[15] Spivey, J.M., *The Z Notation — A Reference Manual* Prentice-Hall, second edition, 1992, out-of-print, available from `http://spivey.oriel.ox.ac.uk/~mike/zrm/index.html`

[16] Sufrin, B. (editor) *Z Handbook* Programming Research Group, Oxford University, March 1986

Index