



B LANGUAGE REFERENCE MANUAL

VERSION 1.8.10

B LANGUAGE
REFERENCE MANUAL

VERSION 1.8.10

Atelier B is a product developed in collaboration with Jean Raymond ABRIAL

Document created by ClearSy

This Document is the property of ClearSy and all reproduction, even partial in any form whatsoever is strictly forbidden without written authorization

All quoted product names are trademarks registered by their respective authors

If you find mistakes or inaccuracies, please contact the maintenance service:

e-mail : maintenance.atelierb@clearsy.com

Tel. : (+33) 4.42.37.12.97

Fax : (+33) 4.42.37.12.71

ClearSy

Support Atelier B

Parc de la Duranne - 320, avenue Archimède

Les Pléiades 3 – Bât A

13857 Aix-en-Provence Cedex 3

FRANCE

CONTENTS

RELEASE NOTES	IV
1. INTRODUCTION	1
2. BASIC CONCEPTS	3
2.1 Lexical Conventions	5
2.2 Syntactic conventions	7
2.3 The DEFINITIONS Clause	8
2.4 Useful Syntax Rules	11
3. TYPING	12
3.1 Typing foundations	12
3.2 B Types	14
3.3 Typing abstract data	15
3.4 Types and constraints of concrete data	16
3.5 Typing of Concrete Constants	20
3.6 Typing of Concrete Variables	23
3.7 Typing operation input parameters	25
3.8 Typing machine parameters	26
3.9 Typing local variables and operation output parameters	27
4. PREDICATES	29
4.1 Propositions	30
4.2 Quantified Predicates	31
4.3 Equality Predicates	32
4.4 Belonging Predicates	33
4.5 Inclusion Predicates	34
4.6 Numbers Comparison Predicates	35
5. EXPRESSIONS	37
5.1 Primary Expressions	38
5.2 Boolean Expressions	39
5.3 Arithmetical Expressions	40
5.4 Arithmetical Expressions (continued)	43
5.5 Expressions of Couples	45
5.6 Building Sets	46
5.7 Set List Expressions	48
5.8 Set List Expressions (continued)	50
5.9 Record expressions	53
5.10 Sets of Relations	55

5.11	Expressions of Relations	56
5.12	Expressions of Relations (continued)	59
5.13	Expressions of Relations (continued)	60
5.14	Expressions of Relations (continued)	61
5.15	Sets of Functions	63
5.16	Expressions of Functions	65
5.17	Sets of Sequences	67
5.18	Sequence Expressions	69
5.19	Sequence Expressions (continued)	71
5.20	Tree sets	73
5.21	Tree Expressions	75
5.22	Tree nodes expressions	78
	Definitions	78
5.23	Binary Tree expressions	80
6.	SUBSTITUTIONS	83
6.1	Block substitution	86
6.2	Identical substitution	87
6.3	Becomes Equal Substitution	88
6.4	Precondition Substitution	90
6.5	Assertion Substitution	91
6.6	Bounded choice Substitution	92
6.7	IF conditional substitution	93
6.8	Conditional Bounded choice Substitution	95
6.9	Case Conditional Substitution	96
6.10	Unbounded choice Substitution	97
6.11	Local Definition Substitution	98
6.12	Becomes Element of Substitution	99
6.13	Becomes such that Substitution	100
6.14	Local Variable Substitution	102
6.15	Sequencing Substitution	103
6.16	Operation Call Substitution	105
6.17	While Loop Substitution	107
6.18	Simultaneous Substitution	109
7.	COMPONENTS	111
7.1	Abstract Machine	111
7.2	Header	113
7.3	Refinement	114
7.4	Implementation	116
7.5	The CONSTRAINTS Clause	118

7.6	The REFINES Clause	119
7.7	The IMPORTS Clause	120
7.8	The SEES Clause	123
7.9	The INCLUDES Clause	127
7.10	The PROMOTES Clause	131
7.11	The EXTENDS Clause	133
7.12	The USES Clause	134
7.13	The SETS Clause	136
7.14	The CONCRETE_CONSTANTS Clause	138
7.15	The ABSTRACT_CONSTANTS Clause	140
7.16	The PROPERTIES Clause	142
7.17	The VALUES Clause	144
7.18	The CONCRETE_VARIABLES Clause	150
7.19	The ABSTRACT_VARIABLES Clause	152
7.20	The INVARIANT Clause	154
7.21	The ASSERTIONS Clause	158
7.22	The INITIALISATION Clause	159
7.23	The OPERATIONS Clause	162
7.24	The LOCAL_OPERATIONS Clause	169
7.25	Specificities of the B0 language	172
	7.25.1 Array controls in B0	172
	7.25.2 TERMS	172
	7.25.3 CONDITIONS	174
	7.25.4 Instructions	175
7.26	Identifier Anti-Collision Rules	177
8. B	ARCHITECTURE	181
8.1	Introduction	181
8.2	B Module	181
8.3	B Project	183
8.4	Libraries	186

RELEASE NOTES

B language Manual Reference, release 1.8.10 (release delivered with Atelier B, release 4.3.0)

1. Add of operators real, ceiling and floor for real numbers.

B language Manual Reference, release 1.8.9 (release delivered with Atelier B, release 4.2.0)

1. Add of real and floating types, and impact on the arithmetic expressions.

B language Manual Reference, release 1.8.7 (commercial release delivered with Atelier B, release 4.0)

Aim: remark integration for Atelier B, release 4.0

1. Removal of closure0 definition.
2. Correction of priorities in symbols table

B language Manual Reference, release 1.8.6 (commercial release delivered with Atelier B, release beta 3.7)

Aim: remark integration for Atelier B, release beta 3.7

1. Removal of all references to partial bijection.

B language Manual Reference, release 1.8.5 (commercial release delivered with Atelier B, release 3.6)

Aim: remark integration for Atelier B, release 3.6

1. Precisions concerning the equivalent model of local operations
2. Restriction concerning the empty set
3. Syntax modifications of relation composition and parallel product in conformity with the existing syntactic analyser
4. Restriction addition for the operation call $x, x \leftarrow op$

B language Manual Reference, release 1.8.4 (commercial release delivered with Atelier B, beta release 3.6)

Aim: User remark integration since the last commercial release 1.8.1

1. Constraints on the literal integers: pointer in the lexical part on a syntactic part

2. Correction to the definition of the mod operator
3. Correction of the exemple concerning the power operator
4. Correction of the typing of the $\text{rel}(R)$ expression
5. Correction of the extension sequence definition
6. Correction of the well defineness of the operators: first, last, front, tail, \uparrow and \downarrow ;
modification of the description of the operators \uparrow and \downarrow
7. Correction of the definition of $\text{rev}(S)$
8. Correction of the definition of \leftarrow
9. Modification of the instruction description «becomes equal to»
10. Correction of the CASE syntax and modification in the order of productions on the
substitutions
11. Correction of the well defineness of $\text{const}(x,q)$ and $\text{infix}(t)$
12. Modification of the cat definition
13. Correction of the example using bin
14. The SET clause is forbidden in a definition
15. Suppression of the restriction on the machine set parameter typing. One need only indicate
that they are types and apply the typing verification rules.
16. Update of priorities of B operators given in appendix, in accordance with the priorities
defined in the Atelier B
17. Correction of the definition of $\text{conc}(S)$
18. Addition of the type of $\text{size}(t)$
19. Addition of grammatical productions on the tree expressions
20. Correction on the binary tree restrictions
21. Correction on the definition of empty Σ or Π
22. Precision in the visibility tables on the non homonymous nature of data associated to a
component
23. Correction on the description of the integer division
24. Modification of the restriction on the parallel call of two operations of an *included*
machine: only modification operation call is forbidden
25. Precision on the trees: trees are never empty
26. Addition of λ in the expression list introducing the quantified variables
27. Modification of the syntax for the expressions using several parameters.
28. Addition in the simple term of bool and access to an array element or a record element in
the arithmetic expressions

29. Modification of the visibility table of local operations in an implementation with regard to its abstraction
30. B operator grouping in the index under the heading ‘#’
31. The effect of deferred set type change, during the valuation in implementation, enlarges on the PROPERTIES clause

B language Manual Reference, release 1.8.3 (Atelier B, internal release 3.6)

Aim: local operation integration for Atelier B, release 3.5

1. Typing: data typing array, operation input parameter typing and operation output parameter typing
2. Simultaneous substitution: modification of the constraint concerning the parallel modification of distinct variables. Restriction on the parallel calls of *imported* operations
3. The OPERATIONS clause: new restrictions. Now this clause contains the implementations of local operations
4. The LOCAL_OPERATIONS clause is a new clause which specifies the local operations
5. Identifier anti-collision controls
6. Atelier B restriction : new restriction on the output parameters with regard to local operations (FT2229)
7. Glossary: modification of operation definitions and addition of local operations
8. Visibility tables of implementation: addition of a new column for the LOCAL_OPERATIONS clause and addition of a new line for the local operations
9. Introduction of the word LOCAL_OPERATIONS as reserved keyword

B language Manual Reference, release 1.8.2 (Atelier B, release 3.5)

Aim: corrections of errors and inaccuracies detected by the Atelier B team. Release 3.5.

1. Precision concerning the using of symbol \$0 in the while loop (refer to section 5.1, *Primary Expressions*, and section 6.17, *While Loop Substitution*)

B language Manual Reference, release 1.8.1 (commercial release delivered with Atelier B, release 3.5)

Aim: correction of the errors detected by the Atelier B team and integration of remarks made by the rereading committee during the meeting number 6 (30/03/98)

1. Corrections concerning the tree integration in the B grammar (index and BNF)
2. Correction concerning the definitions: the body
3. Correction concerning the BNF
4. Corrections of literal string definitions and comments
5. Correction \$0 in the loop ASSERT
6. Suppression of the syntax controls of the presence of the clauses PROPERTIES and INVARIANT
7. Visibility modification of valuations
8. Grammar correction
9. Correction of the definition of \rightsquigarrow
10. Modification of syntax rules
11. Addition of the rule (expression_arithmetic)
12. Suppression of unions of concrete arrays

B language Manual Reference, release 1.8 (commercial release delivered with Atelier B, beta release 3.5)

Aim: evolution of the language supported by the Atelier B, release 3.5 : definition files and records. Trees are not supported by Atelier B release 3.5

1. INTRODUCTION

Preface

The *B Language Reference Manual* describes the B language supported by **Atelier B** release 3.6. This language is based on the language presented in *The B-Book*, however some progresses such as trees, recursivity or multiple refinements are not currently supported by B language.

History

The B method is a formal method enabling the development of secure programs. It was created by Jean-Raymond Abrial, who already during the 1980s, took part in the creation of the \mathbb{Z} notation. G. Laffite, F. Mejia and I. McNeal also contributed to its progressive development. In addition, the B method is based on the scientific work undertaken at Oxford university, in the context of the *Programming Research Group* directed by C.A.R. Hoare. The *B-Book* by J.R. Abrial is the fundamental work that describes the B method.

Aim

The aim of this document is to define the B language precisely, in order to produce a B language reference manual. It is mainly intended for users who perform developments using the B method, and also for all those who wish to discover the possibilities provided by the B language. The language described in this document does not make up a standard itself, however it tends to approach, as closely as possible, the future standard.

Presenting the B method

The development of a project using the B method comprises two activities that are closely linked: *writing* formal texts and *proving* these same texts.

The writing activity comprises writing the specifications for *abstract machines* using high level mathematical formalism that is highly expressive. In this way, a B specification comprises data (that may be expressed among other ways using integers, Boolean values, sets, relations, functions or successions), of *invariable properties* that relate to the data (expressed using logic applied to first order predicates), and finally *services* that allow the initialization and later changes to the data (the transformations to the data are expressed using substitutions). The proof activity for a B specification comprises performing a number of demonstrations in order to prove the establishment and conservation of invariable data properties in the specification (e.g. it is necessary to prove that a service call retains the invariable properties). The generation of proofs to be shown must be completely systematic. It is based especially on the transformation of predicates using substitutions.

The development of an abstract machine continues using an extension to the write activity during the successive *refinement* steps. Refining a specification comprises reformulating it so as to providing it with more and more concrete solutions, but also to enrich it. The proof activity relating to refinements also comprises performing a number of static checks and demonstrations in order to prove that the refinement is a valid reformulation of the specification. The last level of refinement of an abstract machine is called the *implementation*. It is subject to some additional constraints: for example it can

only handle concrete data or substitutions. These constraints make it a programming language that is similar to an imperative language. In this way, it may therefore be run directly on a computer system, using either a dedicated computer or through an intermediate step for automatic translation to Ada, safety Ada or C++.

Reading guide

Chapter 2 *Conventions* presents the principles that apply to the analysis of a text written in B language: lexical analysis, syntax analysis and semantic analysis. It specifies the lexical conventions and describes the different kinds of lexical units. Finally, it presents the syntax conventions used in the rest of the document in order to describe B language grammar.

Chapter 3 *Typing* presents the different forms of data that can be represented in B language, then after introducing types into B, it describes how the typing of data is expressed using typing predicates. Finally, it presents the special case of checking the array type.

Chapter 4 *Predicates* presents the predicate language.

Chapter 5 *Expressions* presents the expression language.

Chapter 6 *Substitutions* presents the substitution language.

Chapter 7 *Components* describes the body of the B components, clause by clause, i.e. need it be reminded, the abstract machines, the refinements and the implementations. It also presents the identifier collision avoidance rules that apply to the components.

Chapter 8 *Architecture* presents the general layout of a B project. It describes the modules, their component (the abstract machines, the refinements and the implementations), the links that exist between the components. Finally, it presents the libraries.

Appendix A *Symbols* presents the table of keywords and the table of operators with their priorities.

Appendix B *Syntax* summarizes B language grammar rules.

Appendix C *Visibility Tables* groups the component visibility rules for a component in relation to the components that it is linked to.

Appendix D presents the *Atelier B V3.6 Restrictions* relating to B language.

Appendix E is the *Glossary*.

Appendix F is the *Index*

2. BASIC CONCEPTS

This chapter presents the general principles of the formal analysis of the B language, along with the lexical and syntactic conventions which have been adopted in the rest of the document.

A B project is made up of a certain number of components (see chapter 7 *Components*). Each component is stored in a separate file. The analysis of a component is split into three successive parts: lexical analysis, syntactic analysis and semantic analysis.

Lexical analysis

Lexical analysis consists in checking that the component is made up of a stream of valid lexems, and in performing the analysis and the replacement of textual definitions (refer to section 2.3 *The Definitions Clause*). On that occasion the elements of the terminal vocabulary of B, such as identifiers, are defined.

Syntactic analysis

Syntactic analysis verifies that the stream of lexems which makes up a component respects the grammatical production rules of B language. These rules are gathered in the appendix B.1.

Semantic Analysis

Lastly, semantic analysis allows checks that the component has a meaning which is in conformity with the B method. In B, semantic analysis can be divided into two phases, a phase of static verification and a phase of proof. The phase of static verification carries out the automatic controls described below:

- **The expressions type check** verifies that the data has been typed correctly and that the expressions used within predicates, other expressions or substitutions, are of compatible types (refer to section 3.1 *Typing Foundations*). These checks are specified in the "Typing Rules" heading for each predicate, expression or substitution.
- **The scope resolution** links each use of an identifier to its definition, i.e. the declaration that defines it. Using the scope rules provided for each predicate, expression or substitution carries out the scope resolution.
- **The visibility check** ensures that the use of a global data in a component clause respects a valid access mode (read or write mode access). The access modes allowed are specified in the visibility tables (refer to *Appendix C*)
- **The identifier collision checks** ensure there is no ambiguity when using data (see chapter 7.26 *Identifier Anti-Collision Rules*).
- **The semantic restrictions** described in the "restrictions" headings in the Predicates, Expressions, Substitutions and Components chapters provides the list of static checks that are not taken into account by the checks described above.

During the proof phase, certain properties for which there is no decision procedure are demonstrated. These properties are called *Proof Obligations*. They are generated systematically from the B components that have already been proved statically. For a B component to be declared correct, all of its Proof Obligations must have been proved by a mathematical demonstration.

In this document, the static semantic checks are described in detail, and the proofs are mentioned but not detailed (refer to the *Proof Obligations Manual*).

2.1 Lexical Conventions

In B language, there are four sorts of lexical units:

- keywords and operators
- identifiers
- literal integers
- literal character strings.

The lexical analysis of a component consists in breaking down its text into a succession of lexems from the beginning of the text, up to the end, while eliminating the superfluous spacing characters and the comments.

The formalism used to describe the lexical units is that of the lexical analyser LEX, which conventions are shown below:

Regular Expression	String	Example
x	The x character	b : the b letter
[x]	The x character	[-] : the minus sign
[xy]	The x or the y character	[bB] : the b letter or the B letter
[x-y]	Any character in the x..y interval, according to ASCII order	[a-z] : a lower case letter
[^x]	Any character but x	[^"] : anything but a quote
x?	x is optional	[~]? 0 : 0 or -0
x*	0 to n occurrences of the x character,	[0-9]* : a positive integer, or nothing
x+	1 to n occurrences of the x character	[0-9]+ : a positive integer
.	Any character but a carriage return	

Here is the description of each type of lexical unit as well as that of the spacing and comment characters.

Keywords and Operators

Keywords and operators are made up of a non-empty sequence of printable characters. Their list is provided in Appendix A. The mathematical symbols used in B Language have their equivalent in ASCII characters. So as to make the reading of this document easier, only the mathematical symbols will be used. The correspondence between the two notations is given in Appendix A.

In order to simplify the syntax of the language, all the operators are given a priority rank as well as an associativity (left or right). Thanks to these two properties, no ambiguity can remain during the syntactic analysis of an expression, or of a predicate made up of several operators.

Identifiers

Ident : [a-zA-Z][a-zA-Z0-9_]*

An identifier is a sequence of letters, figures or of underline characters “_”. The first character must be a letter. Upper and lower case letters are distinguished. An identifier may be of any size.

The dot character "." is not permitted for identifiers. In B language, the dot separates the different renaming prefixes of a renamed component (see chapter 8.3, *Instanciation and renaming*).

Literal Integers

Litéral_integer : [-]? [0-9]⁺

Literal integers are sequences of figures, which can be preceded by a minus sign "-" so as to designate negative integers.

Literal Real

Litéral_real : [-]? [0-9]⁺ [.] [0-9]⁺

Literal reals are numbers with an integer part and a fractional part separated by a dot. They can be prefixed by a minus sign.

Literal character strings

Character_string : [^][.]*

Literal character strings are sequences of characters contained inside quotation marks. All printable ASCII characters are accepted except for the quotation marks "" "" which are used to identify the literal character strings, and the new line character.

Comments

Comments are bounded by the start-of-comment lexem “/*” and the end-of-comment lexem “*/”. The contents of a comment are made of a set of 0 to N printable ASCII characters, with the exception of the two consecutive characters “*/” those build the end-of-comment lexem. In this way, comments can not be nested.

Spacing characters

Spacing characters are the space character ' ', the horizontal and vertical tab characters (HT and VT), the new line characters (CR and FF). The spacing characters are used to separate the lexems. When several spacing characters are used in succession, they are considered like a single space. Spacing characters are necessary to separate a keyword from an identifier. They allow the user complete freedom in choosing the page layout of its B source text.

2.2 Syntactic conventions

The formalism retained for representing the B language syntax is a variation of the BNF and EBNF formalisms, of which the conventions are as follows:

- keywords and operators are represented between quotation marks.
- the other elements of the terminal vocabulary (identifiers, literal integers and literal character strings) are shown in a normal font (neither in italics nor in bold type).
- The non-terminal elements of vocabulary are shown in italics.
- $a ::= b$ represents a grammar production. a is a member of the non-terminal vocabulary, and b is a sequence of concatenated vocabulary items.
- (a) represents the a item
- $a | b$ represents the a item or the b item
- $[a]$ represents the optional a item
- a^* represents n concatenated instances of a , where $n \geq 0$
- a^+ represents n concatenated instances of a , where $n \geq 1$
- a^{*b} represents n concatenated instances of a , separated by b , where $n \geq 0$
- a^{+b} represents n concatenated instances of a , separated by b , where $n \geq 1$

Attention

The characters "() $[\]^+*$ are part of the meta-language of grammar description. They must not be mistaken for operators of B language. The latter are represented between quotation marks, like other keywords and operators of B language.

Example

Clause `abstract_variables ::= "ABSTRACT_VARIABLES" Ident_ren+"`

This grammar production can be used to write the following text:

`ABSTRACT_VARIABLES Var1, instB1.instC2.Var2, instD3.abstr_var`

2.3 The DEFINITIONS Clause

Syntax

The syntactic description of the DEFINITIONS clause uses the BNF notation described in section 2.2 *Syntactic conventions*, and not the LEX notation given previously.

```

Definitions_clause ::= "DEFINITIONS" Definition+ ";"
Definition         ::= Ident [ "(" Ident+ ")" ] "==" Lexem *
                   | "<" Filename ">"
                   | "" " Filename "" "
Definition_call    ::= Ident [ "(" (Lexem+)+ ")" ]

```

The terminal Lexem refers to any lexem among the following lexical units: keywords and operators, identifiers, literal integers and literal character strings (refer to section 2.1 *Lexical Conventions*).

The terminal Filename refers to the name of a file, which can include a relative or absolute path, according to the rules of the operating system with which Atelier B is being used.

Description

The definition clause contains definition files to be included and explicit declarations of textual definitions of a component. The explicit definitions may have parameters. The definition calls located within the text of the component are replaced during the lexical analysis phase, before the syntax analysis. This is the reason why we are presenting the Definitions clause in this chapter. The scope of a definition located in a component is the whole of the component, including the text situated before the declaration of the definition.

Restrictions

1. The various definitions of a component must all have different names.
2. A definition must not use the keywords reserved for the header of a component, or clause names. These are the following keywords: MACHINE, REFINEMENT, IMPLEMENTATION, REFINES, DEFINITIONS, IMPORTS, SEES, INCLUDES, USES, EXTENDS, PROMOTES, SETS, ABSTRACT_CONSTANTS, CONCRETE_CONSTANTS, CONSTANTS, VALUES, ABSTRACT_VARIABLES, VARIABLES, CONCRETE_VARIABLES, INVARIANT, ASSERTIONS, INITIALISATION, OPERATIONS.
3. The optional formal parameters of a definition must all have different names.
4. The operator "==" is illegal on the right hand side of a definition, i.e. in the part located after the "==" operator.
5. The definitions may be dependent on other definitions, provided that there is no cyclic references in their dependencies.
6. When a definition call is made, i.e. when an identifier has the name of a definition outside the definition left part, the name of the definition must be followed by as many effective parameters as there are formal parameters.
7. In the case of the inclusion of a definition file between quotation marks, the name used must designate a file from the current directory, containing the B source file.
8. In the case of the inclusion of a definition file between chevrons (the characters "<" and ">"), the name used must designate a file located in one of the included definitions files

directory.

9. A definitions file must only contain, without taking the commentaries into account, a DEFINITIONS clause respecting the rules described in this paragraph.
10. A definitions file can include other definition files, but these inclusions must not lead to cycles.

USE

A definition is either a reference to a definitions file, or an explicit textual definition. The definitions contained in the definition files are included in the list of definitions of a component as if they were explicit definitions. This allows the sharing of definitions between several components. Indeed, these components need only to include the same definition clause.

If the name of a definition file is between inverted quotation mark, the file is sought from the local directory, where the component being analysed can be found. If the name of a definition file is between chevrons, the file is looked up in order, from each definition directory. This ordered list of directories is supplied by the user of Atelier B.

The name of a definition is an identifier. A definition has parameters if its name is followed by a list of identifiers, which are its formal parameters. The part of a definition located after the "==" operator forms the replacement text for the definition. It is called the definitions body.

The body of a definition is finished when one of the following elements is found: the name of a clause (the list of which is given in Restriction 2), the end of a component, that is to say the word END or a ';' character followed by another definition.

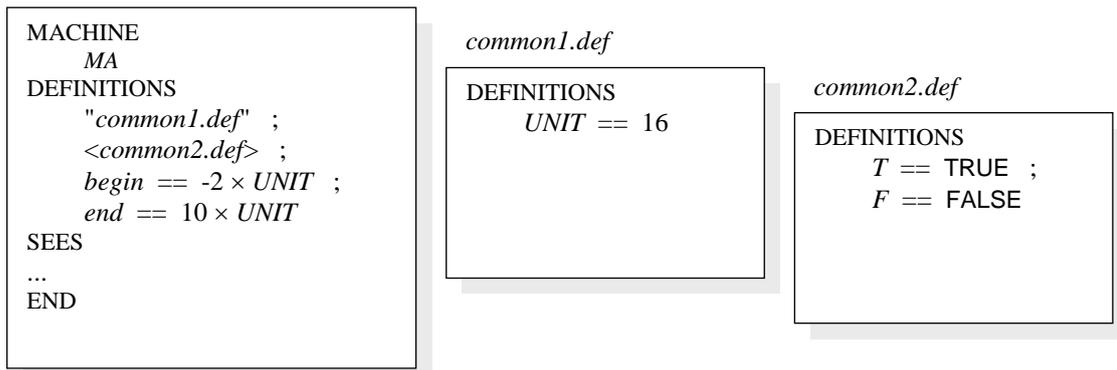
EXAMPLES

```

...
DEFINITIONS
  Composition (f, g) == f ; g ;
  AffectSeq (x, v) == x := 2 × v + 1 ;
CONCRETE_CONSTANTS
...

```

The body of the *Composition* definition is “f ; g”. The last “;” separates this definition from the next one. The body of the *AffectSeq* definition is “x := 2 × v + 1”. The last “;” is part of *AffectSeq* since the DEFINITIONS clause ends as the reserved keyword CONCRETE_CONSTANT is found.



The list of definitions of the component MA is made up of the explicit definitions *begin* and *end* as well as the definitions of files *common1.def* and *common2.def*. The *common1.def* file is looked up from the local directory, whereas the *common2.def* is looked up from the definitions files inclusion directories.

Definition call

A definition call consists of using the definition name, and providing as many effective parameters as the definition has formal parameters.

Once the definition call has been expanded, the usual syntactic rules of component apply.

VISIBILITY

The definitions of a component are local to that component. They are not therefore accessible by the components that are dependent on it. So as to share the definitions, they can be stored in a definition file and be included several times in this file.

EXAMPLES

```

...
DEFINITIONS
  INIT_VAL == -1 ;
  Sum (x1, x2) == ((x1) + (x2)) ;
  INIT_BLOC ==
  BEGIN
    Var1 := INIT_VAL ;
    Var2 := Var1 + 1
  END ;
CONCRETE_CONSTANTS
...

```

Brackets around formal parameters *x1* and *x2* in the *Sum* definition ensure that the sum of *x1* and *x2* will always be performed, even if this definition is called from inside and expression containing operators with a higher priority than '+’.

2.4 Useful Syntax Rules

The following syntax rules are used in the rest of this document in order to simplify the B language syntax.

Syntax

$List_ident$::=	Ident
		"(" Ident ^{+,} " ")"
Ren_ident	::=	Ident ^{+,}

RESTRICTIONS

1. When there is a list of identifiers containing several elements, each identifier must be different.
2. When a renamed identifier is made up of several identifiers, the latter must only be separated by the characters '.', spaces and commentaries are prohibited.

DESCRIPTION

The non-terminal $List_ident$ represents a list from 1 to n identifiers. If the list contains several identifiers, then it must be placed between brackets. Such a list is used to declare data within predicates \forall or \exists or expressions Σ , Π , \cup , \cap or $\{ | \}$.

The non-terminal Ren_Ident represents an identifier which may have been renamed. A renamed identifier has a prefix made up of 1 to n identifiers separated by the dot character. Renamed identifiers designate data from renamed machines (see chapter 5.1 *Primary Expressions*).

EXAMPLES

(x, y, z) is the list of the three pieces of data: x, y, z.

$new.var$ refers to the data var from a renamed machine with the prefix new .

3. TYPING

3.1 Typing foundations

The typing in B is a static checking mechanism of B language data and expressions. The type check of a B component must be done before its proof.

The concept of B type is based on the concept of a set, and the monotony property of inclusion. Let E be an expression, and S and T be sets such that $S \subseteq T$. If $E \in S$ then $E \in T$. The largest set in which E is contained is called the type of E .

In B language, typing is present in three forms: built-in B language types, data typing and type checking.

B language types

The possible types in B language are the basic types and the types built from these using the Cartesian product, the power-set (the set of subsets) and the set of records. This mechanism is described in detail in 3.2.

Data typing

In B language, any data item used in a predicate or in a substitution and that has not yet been assigned a type must be typed. This typing is performed when the data is used for the first time, by scanning the text of the predicate or the substitution from where it starts. Typing is performed using specific predicates or substitutions called typing predicates and typing substitutions, and using a type inference mechanism. The data type is deduced from the predicate or the substitution and from the type of the other data involved. The following sections present the typing predicates that depend on the type of data and the typing substitutions made.

The table below presents for each type of B language data, the clause where it is typed and how it is typed.

Type of data	Typing clause	Typing method
scalar machine parameter (identifier in lowercase)	CONSTRAINTS clause	Typing predicate
Set machine parameter (identifier without lowercase)		Forms a basic type
Deferred or enumerated set	SETS clause	Forms a basic type
Element from an enumerated set	SETS clause	Implicitly typed by the enumerated set
Concrete constant	PROPERTIES clause	Typing predicate for concrete constants
Abstract constant	PROPERTIES clause	Typing predicate for abstract data
Concrete variable	INVARIANT clause	Typing predicate for concrete variable
Abstract variable	INVARIANT clause	Typing predicate for abstract data
Operation input parameter	OPERATIONS clause of abstract machine, in a precondition	Typing predicate for operation input parameter
Operation output parameter	OPERATIONS clause of abstract machine	Typing substitution

Local operation input parameter	LOCAL_OPERATIONS clause, in a precondition	Typing predicate for operation input parameter
Local operation output parameter	LOCAL_OPERATIONS clause	Typing substitution
Predicate variable \forall or \exists , for expression Σ , Π , \cup or \cap , or ANY or LET substitute	Any clause that uses a predicate, an expression or a substitute	Typing the predicate for abstract data
Local variable (VAR substitute)	INITIALISATION or OPERATIONS clause	Typing a substitution

Type checking

Finally, when using data that is already typed in expressions, predicates or substitutions, the typing rules for these expressions, predicates or substitutions must be checked. These rules are provided in the description of each predicate, expression or substitution, in chapters 4, 5 and 6.

3.2 B Types

Syntax

```

Type          ::= Basic_type
               | "P" "(" Type ")"
               | Type x Type
               | "struct" "(" (Ident ":" Type)+ ";" ")"
               | "(" Type ")"

Basic_type    ::= "Z"
               | "R"
               | "FLOAT"
               | "BOOL"
               | "STRING"
               | Ident

```

Description

A B type is either a basic type, or a type built on a type constructor.

The basic types are:

- the set of relative integers \mathbb{Z} ,
- the set of real numbers \mathbb{R} ,
- the set of floating numbers `FLOAT`,
- the set of Boolean `BOOL`, defined as `BOOL = {FALSE, TRUE}`, with `TRUE ≠ FALSE`
- the set of character strings `STRING`,
- the deferred sets and enumerated sets introduced in the `SETS` clause, as well as machine set formal parameters which are considered as deferred sets.

There are three type constructors: the power-set '`P`', the Cartesian product '`×`' and the collection of records labelled '`struct`'. Let $T, T1, T2, T3, T4$ be types:

- $\mathbb{P}(T)$ designates the power-set or the set of subsets of T , i.e. the set the elements of which are sets of elements of T ,
- $T1 \times T2$ designates the Cartesian product of the sets $T1$ and $T2$, i.e. the set of ordered pairs, the first element of which is of the type $T1$ and the second is of the type $T2$. Since the operator '`×`' is associative to the right, the type $T1 \times T2 \times T3 \times T4$ designates the type $((T1 \times T2) \times T3) \times T4$.
- Let n be an integer greater or equal than 1, $T1, \dots, Tn$, be types and $Ident_1 \dots Ident_n$ be identifiers distinct two by two. Then the record type `struct(Ident1:T1, ..., Identn:Tn)` designates the set formed by an ordered collection of n types called record fields. Each field has a name *Ident* called label. These record type labels must be distinct two by two.

EXAMPLES

The type of expression `3` is \mathbb{Z}

The type of expression `{ -5, 3, -1, 8 }` is $\mathbb{P}(\mathbb{Z})$

The type of expression `(0..10) × BOOL → ABS1` is $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \text{BOOL} \times \text{ABS1}))$

The type of expression `rec(a : 5, b : TRUE)` is `struct(a : \mathbb{Z} , b : BOOL)`

3.3 Typing abstract data

Syntax

```

Typing_abstract_data ::=
  Ident "∈" Expression+"x"
  | Ident "⊆" Expression
  | Ident "⊂" Expression
  | Ident+", " "=" Expression+", "

```

Description

The term abstract data applies to an abstract constant, an abstract variable or data introduced by an ANY, LET substitution or by a predicate \forall or \exists , or Σ , Π , \cup , $\{\}$, \cap expression (refer to section 3.1 *Typing foundations*).

The typing predicates for abstract data are specific elementary predicates. Each typing predicate is used to set the type of one or more abstract data elements. It is separated from the preceding and successive predicates by a conjunction.

The elementary typing predicates are belonging, inclusion and equals. The abstract data to type must be in the left hand part of the belonging, inclusion or equals operator. The right hand part is made up of an expression where all of the components are accessible and previously typed (refer to *Typing Order*, below). The type of the abstract data in the left part is then determined by applying the typing rules for the used predicate.

Typing order

The data typing mechanism used within a predicate consists of scanning the entire text of the predicate from start to finish. When a not yet typed data element appears in the left hand side of a typing predicate, the data element is typed and remains so in the rest of the text of the predicate.

Examples

```

VarRaf1 ∈ INT ∧
VarRaf2 ⊆ NAT ∧
VarRaf3 = TRUE ∧
VarRaf4 ∈ (0 .. 5) ↔ (0 .. 10) ∧
VarRaf5 ∈ Z ∧
VarRaf6 ⊆ NAT ∧
VarRaf7 ⊂ NAT1 ∪ (-5 .. -1) ∧
VarRaf8 = (0 .. 4) × {FALSE}

```

As the type of INT is $\mathbb{P}(\mathbb{Z})$ and *VarRaf1* is typed using the operator '∈', the type of *VarRaf1* is \mathbb{Z} .

As the type of NAT is $\mathbb{P}(\mathbb{Z})$ and *VarRaf2* is typed using the operator '⊆', the type of *VarRaf2* is $\mathbb{P}(\mathbb{Z})$.

As the type of TRUE is BOOL and *VarRaf3* is typed using the operator '=', the type of *VarRaf3* is BOOL.

In the same way, the type of *VarRaf4* is $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$, the type of *VarRaf5* is \mathbb{Z} , the type of *VarRaf6* is $\mathbb{P}(\mathbb{Z})$, the type of *VarRaf7* is $\mathbb{P}(\mathbb{Z})$, the type of *VarRaf8* is $\mathbb{P}(\mathbb{Z} \times \text{BOOL})$.

3.4 Types and constraints of concrete data

The concrete data of a B module are data which will be part of the program associated to a module (see section 8.2 *B Module*). Since concrete data must be able to be implemented by a program, a number of constraints have been fixed so as to differentiate concrete data from data which is not concrete. These constraints are necessarily arbitrary, but they have been established by considering what programming languages such as Ada or C++ can easily implement, and by trying to give as much flexibility as possible to B Language users. In this way, integers, booleans or arrays will be able to be implemented (perhaps with certain constraints), but not the data which value is $\{-1, 5, 8\}$ because it is not directly and simply implementable in classical programming languages.

The most important of these constraints is the type of data. For example, a piece of data of type $\mathbb{P}(\mathbb{Z} \times \mathbb{P}(\mathbb{Z}))$ which represents a set of couples the first elements of which are integers, and the second elements are sets of integers is not retained as being concrete because it is too far from what a programming language can implement directly.

There are other constraints than the type. For example, the only concrete data integers are those which are contained between the smallest implementable integer and the largest implementable integer for a given target machine.

Finally, the constraints concerning concrete data depend on the nature of the data. For example, concrete constants can be intervals of integers - but the concrete variables can not. We are first going to describe all the possible categories of concrete data. Then we will present a table giving the authorised categories for each kind of concrete data.

DEFERRED OR ENUMERATED SETS

TYPE

A deferred or enumerated set *Set* is of type $\mathbb{P}(Set)$

CONSTRAINTS

There is no constraint for deferred and enumerated sets.

INTEGERS

TYPE

Concrete integers are of the \mathbb{Z} type.

CONSTRAINTS

Concrete integers must belong to the INT interval whose inferior and superior bounds are the predefined constants MININT and MAXINT. The value of these constants can be parameterized; it depends on the target machine, on which the program associated to a B project will have to work. These values must be placed in a way that any integer comprised between MININT and MAXINT can be directly represented without raising a data overflow on the target machine.

Floating numbers

Type

Floating numbers are of the FLOAT type.

Constraints

There is no constraint.

Booleans**Type**

Booleans are of the BOOL type.

Constraints

There is no constraint.

Elements of abstract or enumerated sets**Type**

The elements belonging to a deferred or enumerated set *Set* are of the type *Set*

Constraints

There is no constraint.

Sub-sets of integers or elements of deferred sets.**TYPE**

Concrete sub-sets of integers are of type $\mathbb{P}(\mathbb{Z})$. Concrete sub-sets of elements of a deferred set *Set* are of type $\mathbb{P}(\text{Set})$

Constraints

Concrete sub-sets of integers must be concrete intervals of integers. Concrete sub-sets of elements of a deferred set must be intervals of concrete integers, when the deferred set is valued by an interval of integers (see section 7.17 The VALUES Clause).

ARRAYS**TYPE**

Concrete arrays are of the type $\mathbb{P}(T_1 \times \dots \times T_n)$, where $n \geq 1$ and each T_i type is a base type other than the STRING type.

Constraints

Before defining the notion of a concrete array, we introduce the notion of simple concrete set. A simple concrete set is a deferred or enumerated set, the boolean values set or an interval of concrete integers or a concrete sub-set of a deferred set.

A concrete array is a total function, of which the original set is the Cartesian product of n simple concrete sets (where $n \geq 1$) and the final set of which is a simple concrete set. The $n-1$ simple sets which make up the domain of definition of the array are also called the index sets of the array.

EXAMPLES

$$\text{Tab1} \in (0..4) \rightarrow \text{INT} \wedge$$

$$\text{Tab2} \in \text{AbstractSet1} \times \text{EnumeratedSet1} \times \text{BOOL} \rightarrow \text{BOOL} \wedge$$

$$\text{Tab3} \in (-1 .. 1) \times \text{IntervalCst1} \rightarrow (0..100) \wedge$$

$$\text{Tab4} \in \text{EnumeratedSet1} \rightarrow \text{NAT} \wedge$$

$Tab5 \in (0..8) \rightarrow INT$

The concrete array *Tab1* is a total function of the interval (0..4) in the INT set. The concrete array *Tab2* is a total surjection of the Cartesian product of the simple sets *AbstractSet1*, *EnumeratedSet1* and BOOL towards the BOOL set.

RECORDS

TYPE

The type of concrete records is defined by induction. A concrete record type is a record type in which field is of one of the following types : \mathbb{Z} , BOOL, a deferred set, an enumerated set, a concrete array type or a concrete record type.

CONSTRAINTS

The constraints on concrete record data are defined by induction. Each field of a concrete record must be a piece of concrete data. More precisely, if one of the fields of a concrete record is itself a record, then each field of the latter must in its turn be a piece of concrete data.

EXAMPLE

```
Year ∈ struct(  Year_number : INT,
               Leap_year   : BOOL,
               Number_of_days : (1..12) → (28..31),
               Weather : struct( Average_Temperature : (1..12) → INT,
                                Average_Rain       : (1..12) → INT ) )
```

The concrete record *Year* contains four fields : the *Year_number* field is a concrete integer, the *Leap_year* field is a Boolean, the *Number_of_days* field is a concrete array and the *Weather* field is a concrete record which has got two concrete array fields, *Average_Temperature* and *Average_Rain*.

CHARACTER STRINGS

TYPE

Character strings are of the STRING type.

CONSTRAINTS

There is no constraint.

Quick reference table

The following table sums up for each piece of concrete data which are the authorised types of concrete data.

Concrete type	Deferred or enumerated set	Integer	Boolean	Item belonging to a deferred or enumerated set	Interval of integers or sub-set of a deferred set	Array	Record	Character string
Nature								
Machine set formal parameter	×				×			
Enumerated or deferred set	×							
Machine scalar formal parameter		×	×	×				
Literal enumerated value				×				
Concrete constant		×	×	×	×	×	×	
Concrete variable		×	×	×		×	×	
Operation input parameter (local or non local)		×	×	×		×	×	×
Operation output parameter (local or non local)		×	×	×		×	×	
Local variable		×	×	×		×	×	

3.5 Typing of Concrete Constants

Syntax

```

Concrete_constant_typing ::=
  Ident+"," "." Typing_belongs_to_concrete_data + "x"
  | Ident "=" Typing_equals_concrete_constant
  | Ident "⊆" Simple_set
  | Ident "⊂" Simple_set

Typing_belongs_to_concrete_data ::=
  Simple_set
  | Simple_set+"x" "→" Simple_set
  | Simple_set+"x" "↘" Simple_set
  | Simple_set+"x" "→" Simple_set
  | Simple_set+"x" "↘" Simple_set
  | "{" Simple_term+"" "}"
  | "struct" "(" (Ident ":" Typing_belongs_to_concrete_data)+" " ")"

Typing_equals_concrete_constant ::=
  Term
  | Array_expr
  | B0_interval
  | B0_Number_set
  | "rec" "(" ( [ Ident ":" ] ( Term | Array_expr )+" " ")"

Simple_set ::=
  B0_simple_set
  | "BOOL"
  | "FLOAT"
  | B0_interval
  | Ident

B0_Simple_set ::=
  "NAT"
  | "NAT1"
  | "INT"

Array_expr ::=
  Ident
  | "{" ( Simple_term+"↦" "↦" Term )+" " "}"
  | Simple_set+"x" "x" "{" Term "}"

B0_interval ::=
  Arithmetic_expression ".." Arithmetic_expression
  | B0_Number_set

```

Description

Concrete constants are typed with the help of typing predicates used in the

PROPERTIES clauses. The typing predicates of concrete constants follow the same principles as the typing predicates of abstract data (refer to section 3.4 *Types and constraints of concrete data.*), while having additional constraints: the right hand side of the predicate must type each constant with a concrete constant type (see 3.4, *Types and constraints of concrete data.*). Only the elementary expressions whose syntax is given below are authorised. They are:

- belonging to a simple set, which is a scalar set.

Example:

$$\begin{aligned} Cst1 &\in \text{INT} \wedge \\ Cst2 &\in \text{BOOL} \wedge \\ Cst3 &\in 0..5 \wedge \\ Cst4 &\in \text{IntervalCst1} \wedge \\ Cst5 &\in \text{AbstractSet1} \wedge \\ Cst6 &\in \text{EnumeratedSet1} \end{aligned}$$

These typing predicates can also be written as follows:

$$Cst1, Cst2, Cst3, Cst4, Cst5, Cst6 \in \text{INT} \times \text{BOOL} \times (0..5) \times \text{IntervalCst1} \times \text{AbstractSet1} \times \text{EnumeratedSet1}$$

- belonging to a set of total functions. The index sets and arrival set of the function must be simple sets.

Example:

$$\begin{aligned} Arr1 &\in (0..4) \rightarrow \text{INT} \wedge \\ Arr2 &\in \text{AbstractSet1} \times \text{EnumeratedSet1} \times \text{BOOL} \twoheadrightarrow \text{BOOL} \wedge \\ Arr3 &\in (-1 .. 1) \times \text{IntervalCst1} \twoheadrightarrow (0..100) \wedge \\ Arr4 &\in \text{EnumeratedSet1} \twoheadrightarrow \text{NAT} \end{aligned}$$

- belonging to a scalar set defined in extension.

Example:

$$\begin{aligned} Cst7 &\in \{ 0, 3, 7, -8 \} \wedge \\ Cst8 &\in \{ \text{bleu}, \text{white}, \text{red} \} \end{aligned}$$

- equality with an identifier which refers to a scalar data.

Example:

$$\begin{aligned} Cst10 &= \text{red} \wedge \\ Cst11 &= Cst10 \end{aligned}$$

- equality with an arithmetic or Boolean expression.

Example:

$$\begin{aligned} Cst12 &= \text{IntCst1} + 2 \times (\text{IntCst2} - 1) \wedge \\ Cst13 &= 0 \wedge \\ Cst14 &= \text{FALSE} \wedge \\ Cst15 &= \text{bool}(Cst12 < 10 \vee Cst12 > 20) \end{aligned}$$

- equality with an array expression

Example:

$$\begin{aligned} Arr5 &= (0..4) \times \{ 0 \} \wedge \\ Arr6 &= (0..2) \times \{ \text{TRUE} \} \end{aligned}$$

- inclusion in a simple set.

Example:

$$\text{IntervCst16} <: \text{INT} \wedge$$
$$\text{IntervCst17} \subseteq \text{AbstractSet1} \wedge$$
$$\text{IntervCst18} \subseteq -100 .. 100 \wedge$$
$$\text{IntervCst19} \subset \text{IntervCst16}$$

- belonging to a record set.

Example:
$$\text{RecCst} \in \text{struct}(\text{masc} : \text{BOOL}, \text{age} : 0.255)$$

3.6 Typing of Concrete Variables

Syntax

$$\begin{aligned}
 \text{Concrete_variable_typing} & ::= \\
 & \text{Ident}^+ \text{,} \text{ " : " } \text{Typing_belongs_to_concrete_data}^+ \text{ }^x \\
 & | \\
 & \text{Ident} \text{ " = " } \text{Term} \\
 \\
 \text{Typing_belongs_to_concrete_data} & ::= \\
 & \text{Simple_set} \\
 & | \\
 & \text{Simple_set}^+ \text{ }^x \text{ " } \rightarrow \text{ " } \text{Simple_set} \\
 & | \\
 & \text{Simple_set}^+ \text{ }^x \text{ " } \succ \text{ " } \text{Simple_set} \\
 & | \\
 & \text{Simple_set}^+ \text{ }^x \text{ " } \rightarrow \text{ " } \text{Simple_set} \\
 & | \\
 & \text{Simple_set}^+ \text{ }^x \text{ " } \succ \text{ " } \text{Simple_set} \\
 & | \\
 & \text{" \{ " } \text{Simple_term}^+ \text{ ,} \text{ " \} " } \\
 & | \\
 & \text{" struct " " (" (Ident " : " Typing_belongs_to_concrete_data)^+ \text{ ,} \text{ ") " }
 \end{aligned}$$

Description

Concrete variables are typed with the help of typing predicates used in the INVARIANT clause. The typing predicates of concrete variables follow the same principles as the typing predicates of abstract data (refer to section 3.4 *Types and constraints of concrete data*), while having additional constraints: only the “belongs to” and “is equal to” predicates are allowed to type concrete variables. Inclusion is forbidden because a concrete variable cannot be a set. Moreover, the right hand side of the predicate must type the variable with a concrete variable type (see section 3.4 *Types and constraints of concrete data*). Only the elementary expressions whose syntax is given below are authorised. They are:

- belonging to a simple set (which is a set of scalars)

Example:

$$\begin{aligned}
 \text{Var1} & \in \text{INT} \wedge \\
 \text{Var2} & \in \text{BOOL} \wedge \\
 \text{Var3} & \in 0..5 \wedge \\
 \text{Var4} & \in \text{IntervalCst1} \wedge \\
 \text{Var5} & \in \text{AbstractSet1} \wedge \\
 \text{Var6} & \in \text{EnumeratedSet1}
 \end{aligned}$$

These typing predicates can also be written as follows :

$$\text{Var1, Var2, Var3, Var4, Var5, Var6} \in \text{INT} \times \text{BOOL} \times (0..5) \times \text{IntervCst1} \times \text{AbstractSet1} \times \text{EnumeratedSet1}$$

- belonging of a set of total functions. The index and arrival sets of the function must be simple sets.

Example:

$$\begin{aligned}
 \text{Arr1} & \in (0..4) \rightarrow \text{INT} \wedge \\
 \text{Arr2} & \in \text{AbstractSet1} \times \text{EnumeratedSet1} \times \text{BOOL} \succ \text{BOOL} \wedge \\
 \text{Arr3} & \in (-1 .. 1) \times \text{IntervalCst1} \succ (0..100) \\
 \text{Arr4} & \in \text{EnumeratedSet1} \succ \text{NAT}
 \end{aligned}$$

- belonging to an extensive set of scalars.

Example:
$$\text{Var7} \in \{ 0, 3, 7, -8 \} \wedge$$
$$\text{Var8} \in \{ \textit{blue}, \textit{white}, \textit{red} \}$$

- equality with an identifier which refers to a scalar data.

Example:
$$\text{Var10} = \textit{red} \wedge$$
$$\text{Var11} = \text{Var10}$$

- equality with an arithmetic or Boolean expression.

Example:
$$\text{Var12} = \textit{IntervalCst1} + 2 \times (\textit{IntervalCst2} - 1) \wedge$$
$$\text{Var13} = 0 \wedge$$
$$\text{Var14} = \text{FALSE} \wedge$$
$$\text{Var15} = \text{bool}(\text{Var12} < 10 \vee \text{Var12} > 20)$$

- belonging to a set of records.

Example:
$$\text{Var16} \in \text{struct}(\text{value} : \text{INT}, \text{status} : \text{BOOL})$$

3.7 Typing operation input parameters

Syntax

```

Operation_input_parameter_typing ::=
  Ident+ "∈" Typing_belongs_to_concrete_data+
  |
  Ident "=" Term

Typing_belongs_to_input_parameter ::=
  Simple_set
  |
  Simple_set+ "→" Simple_set
  |
  Simple_set+ "→" Simple_set
  |
  Simple_set+ "→" Simple_set
  |
  Simple_set+ "→" Simple_set
  |
  "{" Simple_term+ "}"
  |
  "struct" "(" (Ident ":" Typing_belongs_to_concrete_data+ ")"
  |
  "STRING"

```

Description

Operation input parameters are typed inside typing predicates used in a precondition (see section 7.23 *The OPERATIONS Clause*).

There are two different ways of typing operation input parameters, distinguished by the operation belongs to a module which has an associated source code (developed or base module), or if it belongs to an abstract module (see section 8.2 *B Module*).

In the first case, operation input parameters typing predicates follow the same principles as concrete variables typing predicates, while adding another possibility, which is belonging to the set of character strings, STRING. An operation input parameter can therefore be a string of characters.

Example:

```
Var 17 ∈ STRING
```

If the operation belongs to an abstract module, the operation input parameters are abstract data (see section 3.3 *Typing abstract data*). Indeed, since an abstract module does not have an associated code, its operation can not be called from an implementation. It is therefore not necessary for its input parameters to be concrete.

3.8 Typing machine parameters

Syntax

```

Typing_machine_parameter ::=
    Ident+, "∈" Typing_belongs_to_machine_parameter+,
    | Ident+, "=" Term+,

```

```

Typing_belongs_to_machine_parameter ::=
    Number_set
    | "BOOL"
    | Interval
    | Ident

```

```

Number_set ::=
    "Z"
    | "N"
    | "N1"
    | "NAT"
    | "NAT1"
    | "INT"

```

Description

The scalar parameters of an abstract machine are typed with typing predicates in the section 7.5 *The CONSTRAINTS Clause*. The typing predicates of the scalar parameters of a machine follow the same principles as the typing predicates of abstract data.(refer section 3.3 *Typing of abstract data*), to which they add a certain number of restrictions.

The only allowed types to explicitly type the scalar parameters of a machine are \mathbb{Z} , `BOOL` and the set parameters of the abstract machine. The set parameters are exactly the parameters represented by the identifiers without any lower case letter.

The formal scalar parameters that have already been typed in a typing predicate can be used to type another formal scalar parameter.

3.9 Typing local variables and operation output parameters

Description

The local variables declared in a VAR substitution and operation formal output parameters (local or non local) are typed with typing substitutions (see chapter 6 *Substitutions*). Typing substitutions are the following substitutions: “becomes equal to” (see section 6.3), “becomes a member of” (see section 6.12), “become such that” (see section 6.13) and “operation call” (see section 6.16).

As far as operation output parameters are concerned, two cases can be distinguished, according to whether the operation belongs to a module with an associated code (developed module or base module) or to an abstract module (see section 8.2 *B Module*).

In the case of an operation of an abstract module, the operation output parameters are abstract data. Indeed, since an abstract module does not have an associated code, these operations cannot be called by an implantation, and so it is not necessary for the operation output parameters to be concrete.

The output parameters of modules with associated code, as well as implantation local variables must be concrete data. Given a piece of data vI designating a local variable or an untyped operation output parameter (refer to section 2.3 *Order of typing*). Here is how vI can be typed with the help of the different typing substitutions.

Substitution “becomes equal to”

Given an expression E of type T , then the “becomes equal to” substitution: $vI := E$, gives vI the type of T . In order that E may be used to type vI , it is necessary that vI does not appear in E . It is also possible to type vI in a “becomes equal to” substitution in parallel with other data.

Example:

$vI := \text{TRUE}$ the type of vI is **BOOL**
 $vI, v2 := 0, 0$ the types of vI and $v2$ are \mathbb{Z}

Substitution “becomes a member of”

Given an expression E of type $\mathbb{P}(T)$, then the “becomes a member of” substitution $vI : \in E$ gives vI the type of T .

Example:

$vI : \in \text{AbsSet}$ the type of vI is *AbsSet*

Substitution “becomes such that”

Given a predicate P and considering the “becomes such that” substitution: $vI : (P)$, P must type the data vI with an abstract data typing predicate, according to the principles described in section 3.3

Example:

$vI : (vI \in \text{INT} \wedge vI < 10)$ the type of vI is \mathbb{Z}

Substitution “operation call”

Finally, vI can be typed as an operation effective output parameter (during an operation call). The type of vI is then given by the type of the operation (local or non local) formal

output parameter.

Example:

$v1 \leftarrow \text{op1}$

the type of $v1$ is the type of the output parameter of op1

4. PREDICATES

A predicate is a formula that can be proved or disproved or that may be part of the assumptions used to determine proof.

Predicates are used in B language to:

- express the properties of data (within the CONSTRAINTS, PROPERTIES, INVARIANT, ASSERTIONS clauses of the predicates \forall or \exists , expressions λ , $\{ \}$, Σ , Π , \cup or \cap and “becomes such that” substitutions ANY, LET, ASSERT or WHILE),
- express conditions when applying substitutions (SELECT, IF, WHILE substitutions).

The syntax of predicates is given below:

Predicate::=

<i>Bracketed_predicate</i>	Propositions
<i>Conjunction_predicate</i>	
<i>Negation_predicate</i>	
<i>Disjunction_predicate</i>	
<i>Implication_predicate</i>	
<i>Equivalence_predicate</i>	
<i>Universal_predicate</i>	Qualified predicates
<i>Existential_predicate</i>	
<i>Equals_predicate</i>	Equality predicates
<i>Predicate_inequality</i>	
<i>Belongs_predicate</i>	Belonging predicates
<i>Non_belongs_predicate</i>	
<i>Inclusion_predicate</i>	Inclusion predicates
<i>Inclusion_predicate_strict</i>	
<i>Non_inclusion_predicate</i>	
<i>Non_inclusion_predicate_strict</i>	
<i>Less_than_or_equal_predicate</i>	Integer comparison predicates
<i>Strictly_less_than_predicate</i>	
<i>Predicate_greater_than_or_equal</i>	
<i>Strictly_greater_predicate_than</i>	

The following sections describe the predicates grouped in families. For a family of predicates, the predicate operator is given, its syntax in mathematical notation, its type assignment or typing rules and the range of data declared, its description, the applicable laws or mathematical properties and examples.

4.1 Propositions

Operator

()	Brackets
\wedge	Conjunction
\neg	Negation
\vee	Disjunction
\Rightarrow	Implication
\Leftrightarrow	Equivalence

Syntax

<i>Bracketed_predicate</i>	::=	"(" <i>Predicate</i> ")"
<i>Conjunction_predicate</i>	::=	<i>Predicate</i> " \wedge " <i>Predicate</i>
<i>Negation_predicate</i>	::=	" \neg " "(" <i>Predicate</i> ")"
<i>Disjunction_predicate</i>	::=	<i>Predicate</i> " \vee " <i>Predicate</i>
<i>Implication_predicate</i>	::=	<i>Predicate</i> " \Rightarrow " <i>Predicate</i>
<i>Equivalence_predicate</i>	::=	<i>Predicate</i> " \Leftrightarrow " <i>Predicate</i>

Definition

$$P \vee Q = \neg P \Rightarrow Q$$

$$P \Leftrightarrow Q = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

Description

The operators presented allow the construction of complex predicates from simpler predicates. Below, for each complex predicate, is defined exactly when the predicate is true:

Let P and Q be predicates,

- (P) is true if and only if P is true. This construction allows placing predicates in brackets, which may become necessary depending on the priority level of the operators used.
For example, the $P \wedge Q \Rightarrow R$ predicate will be analyzed as $(P \wedge Q) \Rightarrow R$ and not as $P \wedge (Q \Rightarrow R)$, as the ' \wedge ' operator has a higher priority than the ' \Rightarrow ' operator. To express the $P \wedge (Q \Rightarrow R)$ predicate, brackets are therefore an absolute necessity.
- $P \wedge Q$ is true if and only if P and Q are true,
- $\neg(P)$ is true if and only if P is not true,
- $P \vee Q$ is true if and only if P or Q is true,
- $P \Rightarrow Q$ is true if and only if Q is true or P is not true,
- $P \Leftrightarrow Q$ is true if and only if $P \Rightarrow Q$ and $Q \Rightarrow P$ are true.

4.2 Quantified Predicates

Operator

\forall	Universal quantifier
\exists	Existential quantifier

Syntax

Universal_predicate ::= " \forall " *List_ident* "." "(" *Predicate* " \Rightarrow " *Predicate* ")"
Existential_predicate ::= " \exists " *List_ident* "." "(" *Predicate* ")"

Definition

$$\exists X.(P) = \neg(\forall X.(\neg(P)))$$

Scope

Predicates $\forall X.(P)$ and $\exists X.(P)$ introduce a list of data X , the scope of which is predicate P .

Restrictions

- The variables introduced by universal predicates, format $\forall X.(P \Rightarrow Q)$ must be typed by an abstract data typing predicate (refer to 3.1 Typing foundations), taken from a list of conjunctions at the highest level of syntax analysis in P .
- The variables introduced by an existence predicate, format $\exists X.(P)$ must be typed by an abstract data typing predicate (refer to 3.1 Typing foundations), taken from a list of conjunctions at the highest level of syntax analysis in P .

Description

Let X be a list of identifiers that are distinct when taken two by two and P and Q predicates.

- The $\forall X.(P \Rightarrow Q)$ predicate is true if predicate $P \Rightarrow Q$ is true regardless of the values of X .
- The $\exists X.(P)$ predicate is true if a non empty value set exists for X where predicate P is true.

Examples

Let A be the set of integers: $A = \{ 0, 1, 2 \}$.

Predicate $\forall x.(x \in A \Rightarrow x \leq 2)$ is true as any element of A is less than or equal to 2.

Predicate $\exists x.(x \in A \wedge x = 2)$ is true as an element of A exists that is equal to 2.

4.3 Equality Predicates

Operator

=	Equal to
≠	Unequal to

Syntax

<i>Equals_predicate</i>	::=	<i>Expression</i> "=" <i>Expression</i>
<i>Predicate_unequal</i>	::=	<i>Expression</i> "≠" <i>Expression</i>

Typing rule

In the $x = y$ and $x \neq y$ predicates, the expressions x and y must have the same type.

Definition

$$x \neq y = \neg (x = y)$$

Description

- Predicate $x = y$ is true if the expressions x and y have the same value.
- Predicate $x \neq y$ is true if the expressions x and y , although of the same type, do not have the same value.

4.4 Belonging Predicates

Operator

\in	Belonging
\notin	Non belonging

Syntax

<i>Belongs_predicate</i>	::=	<i>Expression</i> " \in " <i>Expression</i>
<i>Non_belongs_predicate</i>	::=	<i>Expression</i> " \notin " <i>Expression</i>

Typing rule

In predicates $x \in E$ and $x \notin E$, if the type of expression x is T then the type for E must be $\mathbb{P}(T)$.

Definition

$$x \notin E = \neg(x \in E)$$

Description

Let x and E be expressions.

- Predicate $x \in E$ is true if the value of expression x belongs to set E .
- Predicate $x \notin E$ is true if the value of expression x does not belong to set E .

4.5 Inclusion Predicates

Operator

\subseteq	Inclusion
\subset	Strict inclusion
$\not\subseteq$	Non inclusion
$\not\subset$	Strict non inclusion

Syntax

<i>Inclusion_predicate</i>	::=	<i>Expression</i> " \subseteq " <i>Expression</i>
<i>Strict_inclusion_predicate</i>	::=	<i>Expression</i> " \subset " <i>Expression</i>
<i>Non_inclusion_predicate</i>	::=	<i>Expression</i> " $\not\subseteq$ " <i>Expression</i>
<i>Predicate_strict_non_inclusion</i>	::=	<i>Expression</i> " $\not\subset$ " <i>Expression</i>

Definition

$$s \subseteq T = s \in \mathbb{P}(T)$$

$$s \subset T = s \subseteq T \wedge s \neq T$$

$$s \not\subseteq T = \neg(T \in \mathbb{P}(T))$$

$$s \not\subset T = \neg(s \subseteq T \wedge s \neq T)$$

Typing rule

In predicates $X \subseteq Y$, $X \subset Y$, $X \not\subseteq Y$, $X \not\subset Y$, expressions X and Y must have the same type, and their type must match $\mathbb{P}(T)$.

Description

Let X and Y be expressions representing sets.

- $X \subseteq Y$ is true if any element of X belongs to Y .
- $X \subset Y$ is true if any element of X belongs to Y and if X and Y are different.
- $X \not\subseteq Y$ is true if an element of X exists that does not belong to Y .
- $X \not\subset Y$ is true if X is equal to Y or if an element of X exists that does not belong to Y .

4.6 Numbers Comparison Predicates

Operator

\leq	Less than or equal to
$<$	Strictly less than
\geq	Greater than or equal to
$>$	Strictly greater than

Syntax

<i>Less_than_or_equal_predicate</i>	::=	<i>Expression</i> " \leq " <i>Expression</i>
<i>Strictly_less_than_predicate</i>	::=	<i>Expression</i> " $<$ " <i>Expression</i>
<i>Predicate_greater_than_or_equal</i>	::=	<i>Expression</i> " \geq " <i>Expression</i>
<i>Strictly_greater_predicate_than</i>	::=	<i>Expression</i> " $>$ " <i>Expression</i>

Typing rule

In predicates $x \leq y$, $x < y$, $x \geq y$, $x > y$, the expressions x and y must be of type \mathbb{Z} , \mathbb{R} or `FLOAT`.

Description

Let x and y be expressions representing integers:

- $x \leq y$ is true if x is less than or equal to y ,
- $x < y$ is true if x is strictly less than y ,
- $x \geq y$ is true if x is greater than or equal to y ,
- $x > y$ is true if x is strictly greater than y .

5. EXPRESSIONS

Syntax

The syntax of expressions is shown below:

```
Expression ::=
    Expressions_primary
    | Expressions_boolean
    | Expressions_arithmetical
    | Expressions_of_couples
    | Expressions_of_sets
    | Construction_of_sets
    | Expressions_of_records
    | Expressions_of_relations
    | Expressions_of_functionss
    | Construction_of_functionss
    | Expressions_of_sequences
    | Construction_of_sequences
    | Expressions_of_trees
```

Description

An expression is a formula that designates a data item. An expression takes a value that belongs to a B language type.

The following sections describe the expressions gathered by families. For a family of expressions, the name of the expression is given, along with their syntax in mathematical notation, their typing rules, the scope of the data they declare, their definition, their well defineness, their description and the appropriate examples.

5.1 Primary Expressions

Operator

	Data
.	Renaming data
\$0	The specific value of a data item
()	Expression in brackets
" "	Character string

Syntax

<i>Data</i>	::=	Ident_ren Ident_ren"\$0"
<i>Expr_in_brackets</i>	::=	(" Expression ")
<i>String_lit</i>	::=	String_of_character

Typing rules

- Let d be the name at a type T data item and r a renaming prefix. Then, the type of $r.d$, $d\$0$ and $r.d\$0$ is T .
- Let E be a type T expression, the type of (E) is T .
- A literal character string type is STRING

Description

The d expression designates a data item defined in a B component. This may be a formal component parameter, a deferred or enumerated set, an element in an enumerated set, a variable, a constant, a formal operation parameter, quantified data item (introduced by a quantification predicate or by an ANY or LET substitution) or a local variable (introduced by a VAR substitution). When the data name has prefixes, these express the successive renamings of d (refer to section 8.3 *Instantiating and Renaming*).

With data d . The primary expression $d\$0$ can only be encountered in two cases:

- In the predicate of a substitution “becomes such that”, $d\$0$ refers to the value of variable d before the application of the substitution,
- In the predicate of an ASSERT or WHILE substitution $d\$0$ refers to the value of variable d in the abstraction.

An expression in brackets is equivalent to the internal expression. It is used to group an expression. As the analysis of an expression depends on the priority level and on the associativity of the operators, the use of brackets is sometimes required to represent certain expressions.

Examples

Data names: $x1$, $Monday$, nbr_of_days , $a1.b1.CTS_START$

Data names before substitutions, or data for the abstraction: $x1\$0$, $cc_02.var\$0$

Expressions in brackets: $(x + y) \times z$, the expression $x + y$ must be in brackets as operator ‘ \times ’ has a higher priority than operator ‘+’.

Literal character string : “Hello world!”

5.2 Boolean Expressions

Operator

TRUE	value of true
FALSE	value of false
bool	conversion of a predicate into a Boolean expression

Syntax

```

Boolean_lit ::= "FALSE"
              | "TRUE"
Conversion_Boolean ::= "bool" "(" Predicate ")"

```

Typing rule

The type of boolean expressions is BOOL.

Definition

$BOOL = \{ FALSE, TRUE \}$

Description

- TRUE and FALSE are literal constants of the predefined BOOL set (refer to 3.1 Typing foundations). By convention, constant TRUE refers to the boolean value of true and the constant FALSE refers to the boolean value of false.
- The bool operator is used to convert a predicate into a boolean expression. Let P be a predicate, the expression $Bool(P)$ takes the value TRUE if P is true and FALSE if not.

Example

Expression: $bool(\exists x. (x \in \mathbb{N}_1 \wedge x = x^2))$ takes as its value TRUE.

Expression: $bool(b = TRUE)$ takes as its value b .

5.3 Arithmetical Expressions

Operator

MAXINT	The largest implementable integer
MININT	The smallest implementable integer
+	Addition
-	Difference, and also unary minus
×	Product
/	Integer division
mod	Modulo
x^y	Power of
succ	Successor
pred	Predecessor
floor	Floor function
ceiling	Ceiling function
real	Conversion from \mathbb{Z} to \mathbb{R}

Syntax

<i>Integer_lit</i>	::=	Literal_Integer "MAXINT" "MININT"
<i>Addition</i>	::=	<i>Expression</i> "+" <i>Expression</i>
<i>Difference</i>	::=	<i>Expression</i> "-" <i>Expression</i>
<i>Unary_Minus</i>	::=	"-" <i>Expression</i>
<i>Product</i>	::=	<i>Expression</i> "×" <i>Expression</i>
<i>Division</i>	::=	<i>Expression</i> "/" <i>Expression</i>
<i>Modulo</i>	::=	<i>Expression</i> "mod" <i>Expression</i>
<i>Puissance</i>	::=	<i>Expression</i> ^{<i>Expression</i>}
<i>Successor</i>	::=	"succ" ["(" <i>Expression</i> ")"]
<i>Predecessor</i>	::=	"pred" ["(" <i>Expression</i> ")"]
<i>Floor</i>	::=	"floor" "(" <i>Expression</i> ")"
<i>Ceiling</i>	::=	"ceiling" "(" <i>Expression</i> ")"
<i>Real_conversion</i>	::=	"real" "(" <i>Expression</i> ")"

Typing rule

The type of the literal integers and the predefined constants MAXINT and MINTINT is \mathbb{Z} .

In expressions: $x + y$, $x - y$, $-x$, $x \times y$ and x / y , expressions x and y must be both of the type \mathbb{Z} , \mathbb{R} or FLOAT . In expressions: $x \bmod y$, x^y , $\text{succ}(x)$ and $\text{pred}(x)$, expressions x and y must be type \mathbb{Z} ones. The type of these expressions is the same as the operand. The type of the successor and predecessor functions is $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$.

In $\text{floor}(x)$ and $\text{ceiling}(x)$ expressions, x must be of the type \mathbb{R} . The type of floor and ceiling functions is $\mathbb{P}(\mathbb{R} \times \mathbb{Z})$.

In the expression $\text{real}(x)$, x must be of the type \mathbb{Z} . The type of real function is $\mathbb{P}(\mathbb{Z} \times \mathbb{R})$.

Well defineness

Expression	Well defineness condition
a / b	$b \in \mathbb{Z} - \{0\}$
$a \bmod b$	$a \in \mathbb{N} \wedge b \in \mathbb{N}_1$
a^b	$a \in \mathbb{Z} \wedge b \in \mathbb{N}$

Description

Let x and y be integer type expressions, then:

- The predefined constants MAXINT and MININT respectively represent the largest and the smallest concrete integer that can be used in B. The literal integers in B language must be in the range between MAXINT and MININT. The values of MAXINT and MININT are set for a given project according to the target machine that the program will be run on. If integers are stored in a specific machine using four bytes, then the values of MAXINT and MININT may be -2^{31} and $2^{31}-1$.
- $x + y$ represents the sum of x and y .
- $x - y$ represents the difference between x and y .
- $-x$ represents the opposite of x .
- $x \times y$ represents the multiplication of x by y .
- x / y with x and y some integers, represents the integer division of x by y . For the integer division to be meaningful, it is necessary that $y \neq 0$. The integer division is defined as follows: with $x \in \mathbb{N}$ and $y \in \mathbb{N}_1$, then $x / y = \max(\{q \mid q \in \mathbb{N} \wedge y \times q \leq x\})$. In the same way it is possible to define the following constraints: if $q = x / y$ then $q \times y \leq x \wedge x < (q + 1) \times y \wedge q \geq 0$.

Then this definition is extended to relative integers, thanks to the signs rule (refer to *laws*).

x / y with x and y some real or floating number, represents the inverse of the multiplication.

- $x \bmod y$ represents the remainder of the integer division of x by y . The modulo operator is only defined for values of x belonging to \mathbb{N} and for values of y belonging to \mathbb{N}_1 .
- If $x \in \mathbb{Z}$ and $y \in \mathbb{N}$, then x^y represents x to the integer power of y .
- succ represents the successor function, defined for \mathbb{Z} in \mathbb{Z} . $\text{succ}(x)$ represents the successor of x , i.e. $x + 1$.
- pred represents the predecessor function, defined for \mathbb{Z} in \mathbb{Z} . $\text{pred}(x)$ represents the predecessor of x , i.e. $x - 1$.
- real represents the conversion function from $\mathbb{Z} \vee \mathbb{R}$. $\text{real}(x)$ is the real z such that $z = x$.

Let x be a real type expression, then:

- floor represents the floor function, defined for \mathbb{R} in \mathbb{Z} . $\text{floor}(x)$ represents the floor of x , i.e. the only integer n such that $n \leq x < n+1$.
- ceiling represents the ceiling function, defined for \mathbb{R} in \mathbb{Z} . $\text{ceiling}(x)$ represents the ceiling of x , i.e. the only integer n such that $n-1 < x \leq n$.

Laws

With $x \in \mathbb{N}$ then $x0 = 1$

With $x \in \mathbb{N}$ and $y \in \mathbb{N} - \{0\}$, then:

$$(-x) / y = -(x / y)$$

$$x / (-y) = -(x / y)$$

$$x \text{ mod } y = x - y \times (x / y)$$

Examples

$$b^2 - 4 \times a \times c$$

$$x - x^3 / 6 + x^5 / 120$$

$$x \times a + y^2 + z \text{ mod } 9 - 7$$

5.4 Arithmetical Expressions (continued)

Operator

max	Maximum
min	Minimum
card	Cardinal
Σ	Sum of arithmetical expressions
Π	Product of arithmetical expressions

Syntax

<i>Maximum</i>	::=	"max" "(" <i>Expression</i> ")"
<i>Minimum</i>	::=	"min" "(" <i>Expression</i> ")"
<i>Cardinal</i>	::=	"card" "(" <i>Expression</i> ")"
<i>Generalized_sum</i>	::=	" Σ " <i>List_ident</i> "." "(" <i>Predicate</i> " " <i>Expression</i> ")"
<i>Produit_generalized</i>	::=	" Π " <i>List_ident</i> "." "(" <i>Predicate</i> " " <i>Expression</i> ")"

Typing rule

The type of the arithmetic expressions presented above is \mathbb{Z} or \mathbb{R} (depending of the type of the operand).

In the expressions : $\max(E)$, $\min(E)$, E must be a set of integers, of type $\mathbb{P}(\mathbb{Z})$ or $\mathbb{P}(\mathbb{R})$.

In the expression : $\text{card}(E)$, E must be a set, of type $\mathbb{P}(T)$.

In the expressions : $\Sigma X.(P|E)$, $\Pi X.(P|E)$, the E expressions must be of type \mathbb{Z} or \mathbb{R} .

Well defineness

Expression	Well defineness condition
$\max(E)$	E must be non empty and must have an upper bound
$\min(E)$	E must be non empty and must have a lower bound
$\text{card}(E)$	E must be finite
$\Sigma x.(P E)$	the $\{x P\}$ set must be finite
$\Pi x.(P E)$	

Scope

In the expressions: $\Sigma X.(P|E)$, $\Pi X.(P|E)$, the scope of the list of identifiers X is predicate P and expression E .

Restrictions

- The variables introduced by the expressions $\Sigma X.(P|E)$ or $\Pi X.(P|E)$ must be typed by an abstract data typing predicate (refer to 3.1 *Typing foundations*), found in a list of conjunctions at the highest syntax analysis level for P . These variables cannot be used in P before they have been typed.

Description

Let E be an expression representing a non empty set of numbers.

- $\max(E)$ represents the largest element in E and $\min(E)$ represents the smallest element in E .

Let F be an expression representing a finite set.

- $\text{card}(F)$ represents the number of elements in F .

Let X be a list of names of variables that are distinct two by two. Let P be a predicate that types the variables in the list X and E an integer type expression.

- $\Sigma(X).(P|E)$ represents the sum of expressions E corresponding to values of the X variables that establish P . If $\{X|P\} = \emptyset$ then the sum equals 0.
- $\Pi(X).(P|E)$ represents the product of the expressions E that correspond to the values of the X variables the establish P . If $\{X|P\} = \emptyset$ then the product equals 1.

In cases where the list of variables only comprises one name x , the following syntax is used $\Sigma x.(P|E)$ and $\Pi x.(P|E)$.

Examples

With $E = \{-1, 2, 9, -6\}$,

$$\max(E) = 9 \text{ and } \min(E) = -6$$

With $FRUITS = \{Strawberry, Blueberry, Raspberry\}$,

$$\text{card}(FRUITS) = 3$$

$$\Sigma x.(x \in \{1, 2, 3\} | x + 1) = (1 + 1) + (2 + 1) + (3 + 1) = 9$$

$$\Pi x.(x \in \mathbb{N}_1 \wedge x \leq 3 | x) = 1 \times 2 \times 3 = 6$$

5.5 Expressions of Couples

Operator

\mapsto Binary correspondence (maplet)

Syntax

$$\begin{array}{l} \text{Couple} ::= \text{Expression } \mapsto \text{Expression} \\ | \text{Expression } \text{" , " } \text{Expression} \end{array}$$

Typing rule

If x and y are respectively types T and U , then $x \mapsto y$ is type $T \times U$.

Description

A couple is an ordered pair of elements, its notation is $(x \mapsto y)$. A relation R of a set E in a set F is a set of couples $(x \mapsto y)$ where x belongs to E and y belongs to F . If $(i \mapsto j)$ is an element of a relation R , it is said that j is associated with i by R . As the relations are sets, all of the operators on the sets may be applied to relations.

Examples

$$rel1 = \{(0 \mapsto \text{FALSE}), (1 \mapsto \text{TRUE}), (2 \mapsto \text{FALSE}), (3 \mapsto \text{TRUE}), (4 \mapsto \text{FALSE}), (5 \mapsto \text{TRUE})\}$$

$$rel2 = \{((0 \mapsto \text{FALSE}) \mapsto 7), ((0 \mapsto \text{TRUE}) \mapsto 9), ((1 \mapsto \text{FALSE}) \mapsto 6), ((1 \mapsto \text{TRUE}) \mapsto 8)\}$$

$rel1$ is a relation of $0..5$ to **BOOL** and $rel2$ is a relation of $\{0, 1\} \times \text{BOOL}$ to $6..9$.

5.6 Building Sets

Operator

\emptyset	Empty-set
\mathbb{Z}	Set of relative integers
\mathbb{N}	Set of integers
\mathbb{N}_1	Set of non null integers
NAT	Set of implementable integers
NAT ₁	Set of non null implementable integers
INT	Set of implementable relative integers
\mathbb{R}	Set of real numbers
FLOAT	Set of floating numbers
BOOL	Set of Boolean values
STRING	Set of character strings

Syntax

<i>Empty_set</i>	::=	" \emptyset "
<i>Number_set</i>	::=	" \mathbb{Z} "
		" \mathbb{N} "
		" \mathbb{N}_1 "
		"NAT"
		"NAT ₁ "
		"INT"
		" \mathbb{R} "
		"FLOAT"
<i>Boolean_set</i>	::=	"BOOL"
<i>Strings_set</i>	::=	"STRING"

Typing rule

The empty set \emptyset has no fixed type set. It may take the same type as any other set, depending on the context that it is used in. Its type is matching $\mathbb{P}(T)$.

Type of sets \mathbb{Z} , \mathbb{N} , \mathbb{N}_1 , NAT, NAT₁ and INT is $\mathbb{P}(\mathbb{Z})$.

Type of set \mathbb{R} is $\mathbb{P}(\mathbb{R})$

Type of set FLOAT is $\mathbb{P}(\text{FLOAT})$.

Type of set BOOL is $\mathbb{P}(\text{BOOL})$.

Type of set STRING is $\mathbb{P}(\text{STRING})$.

Restriction

When using the empty set \emptyset (in a predicate, an expression or a substitution), the type of the empty set must be given according with the context.

For example, the predicate $\emptyset = \emptyset$ is forbidden. The substitution $x := \emptyset$ is valid only if x has already been typed.

Definitions

$\text{NAT} = 0 \dots \text{MAXINT}$

$\text{NAT}_1 = \text{NAT} - \{0\}$

$\text{INT} = \text{MININT} \dots \text{MAXINT}$

$\text{BOOL} = \{\text{FALSE}, \text{TRUE}\}$

Description

- The empty set \emptyset is a set that does not have any elements. It may be defined as the difference between any set and itself, which explains why it can take the same type as any set type.
- Set \mathbb{Z} refers to the set of relative integers. Set \mathbb{N} refers to the set of natural integers. Set \mathbb{N}_1 refers to the set of strictly positive natural integers.
- Set INT refers to the set of implementable relative integers.
- Set NAT refers to the set of natural implementable integers.
- Set NAT_1 refers to the set of strictly positive natural implementable integers.
- Set \mathbb{R} is the set of real number.
- Set FLOAT is the set of floating number.
- Set BOOL refers to the set of Boolean values.
- Set STRING refers to the set of character strings.

5.7 Set List Expressions

Operator

\times	Cartesian product
\mathbb{P}	Set of sub-sets (power-set)
\mathbb{P}_1	Set of non empty sub-sets
\mathbb{F}	Set of finite sub-sets
\mathbb{F}_1	Set of non empty finite sub-sets
$\{ \}$	Set defined in comprehension
$\{ \}$	Set defined in extension
$..$	Interval

Syntax

<i>Product</i>	::=	<i>Expression</i> "×" <i>Expression</i>
<i>Comprehension_set</i>	::=	"{" Ident ⁺ " " Predicate "}"
<i>Subsets</i>	::=	" \mathbb{P} " "(" <i>Expression</i> ")" " \mathbb{P}_1 " "(" <i>Expression</i> ")"
<i>Finite_subsets</i>	::=	" \mathbb{F} " "(" <i>Expression</i> ")" " \mathbb{F}_1 " "(" <i>Expression</i> ")"
<i>Set_extension</i>	::=	"{" <i>Expression</i> ⁺ "}"
<i>Interval</i>	::=	<i>Expression</i> ".." <i>Expression</i>

Definitions

$$\mathbb{P}_1(E) = \{ F \mid F \in \mathbb{P}(E) \wedge F \neq \{ \emptyset \} \}$$

$$\mathbb{F}_1(E) = \{ F \mid F \in \mathbb{F}(E) \wedge F \neq \{ \emptyset \} \}$$

Typing rule

Let X be a set $\mathbb{P}(T1)$ type expression and Y a $\mathbb{P}(T2)$ set type expression. The type of $X \times Y$ is $\mathbb{P}(T1 \times T2)$. The type of $\mathbb{P}(X)$, $\mathbb{P}_1(X)$, $\mathbb{F}(X)$ and $\mathbb{F}_1(X)$ is $\mathbb{P}(\mathbb{P}(T1))$.

Let $E1, \dots, En$ be expressions of the same type T , then the type of $\{ E1, \dots, En \}$ is $\mathbb{P}(T)$.

Let X be a list of identifiers that are distinct taken two by two $x1, \dots, xn$ typed in predicate P and with types $T1, \dots, Tn$. Then, the type of the comprehension set $\{ X \mid P \}$ is $\mathbb{P}(T1 \times \dots \times Tn)$. In cases where X comprises only one identifier, the type of $\{ X \mid P \}$ is $\mathbb{P}(T1)$.

Let X and Y be integer \mathbb{Z} type expressions, then the type of $X..Y$ is $\mathbb{P}(\mathbb{Z})$.

Scope

Let X be a list of identifiers and P a predicate, then the comprehension set is $\{ X \mid P \}$, the scope of identifiers in list X is predicate P .

Restrictions

- The X variables introduced by the expressions matching $\{ X \mid P \}$ must be distinct two by two
- The X variables introduced by the expressions matching $\{ X \mid P \}$ must be typed by

an abstract data typing predicate (refer to 3.1 *Typing foundations*), located in a list of conjunctions at the highest syntax analysis level in P . These variables cannot be used in P before they have been typed.

Description

- Let $E1$ and $E2$ be sets, then $E1 \times E2$ refers to the Cartesian product of $E1$ and $E2$, i.e. the set of couples where the first element belongs to $E1$ and the second element belongs to $E2$. If $E1$, $E2$ and $E3$ are sets, $E1 \times E2 \times E3$ and $(E1 \times E2) \times E3$ designated the set of couples $((x1 \mapsto x2) \mapsto x3)$, and $E1 \times (E2 \times E3)$ refers to the set of couples $(x1 \mapsto (x2 \mapsto x3))$.
- Let $x1, \dots, xn$ be expressions, then the extension set $\{x1, \dots, xn\}$ represents the set with elements $x1, \dots, xn$.
- Let $E1$ be a set, then $\mathbb{P}(E1)$ represents the set of parts of $E1$. $\mathbb{P}1(E1)$ represents the set of non empty parts of $E1$. $\mathbb{F}(E1)$ represents the set of finite parts in $E1$. $\mathbb{F}1(E1)$ represents the set of non empty finite parts in $E1$.
- Let X be a list of identifiers $x1, \dots, xn$ and P a predicate that types X and expresses the properties on X . Then, the set in comprehension $\{X|P\}$ represents the set of maplets $(x1 \mapsto \dots \mapsto xn)$ that verify P . In cases where X comprises only one identifier, $\{X|P\}$ represents the set of $x1$ elements that verify P .
- Let $x1$ and $x2$ be integers, then the interval $x1 \dots x2$ represents the set of integers that are greater than or equal to $x1$ and less than or equal to $x2$. Therefore in cases where $x1 > x2$, $x1 \dots x2$ represents the empty set.

Examples

Let $E1$ and $E2$ be sets defined in extension by: $E1 = \{1, 2, 3\}$ and $E2 = \{4, 5\}$

$$E1 \times E2 = \{(1 \mapsto 4), (1 \mapsto 5), (2 \mapsto 4), (2 \mapsto 5), (3 \mapsto 4), (3 \mapsto 5)\}$$

$$\{x1 \mid x1 \in E1 \wedge x1 \bmod 2 = 1\} = \{1, 3\}$$

$$\{x1, x2 \mid x1 \in \mathbb{N} \wedge x2 \in \mathbb{N} \wedge x1 < x2 \wedge x2 < 3\} = \{(0 \mapsto 1), (0 \mapsto 2), (1 \mapsto 2)\}$$

$$\mathbb{P}(E1) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

$$\mathbb{P}1(E1) = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

$$\mathbb{F}(E1) = \mathbb{P}(E1)$$

$$\mathbb{F}1(E1) = \mathbb{P}1(E1)$$

$$-1..5 = \{-1, 0, 1, 2, 3, 4, 5\}$$

$$6..4 = \emptyset$$

5.8 Set List Expressions (continued)

Operator

-	Difference
\cup	Union
\cap	Intersection
union	Generalized union
inter	Generalized intersection
\bigcup	Quantified union
\bigcap	Quantified intersection

Syntax

<i>Difference</i>	::=	<i>Expression</i> "-" <i>Expression</i>
<i>Union</i>	::=	<i>Expression</i> " \cup " <i>Expression</i>
<i>Intersection</i>	::=	<i>Expression</i> " \cap " <i>Expression</i>
<i>Generalized_union</i>	::=	"union" "(" <i>Expression</i> ")"
<i>Generalized_intersection</i>	::=	"inter" "(" <i>Expression</i> ")"
<i>Quantified_union</i>	::=	" \bigcup " <i>List_ident</i> "." "(" <i>Predicate</i> ^{+"^"} " " <i>Expression</i> ")"
<i>Quantified_intersection</i>	::=	" \bigcap " <i>List_ident</i> "." "(" <i>Predicate</i> ^{+"^"} " " <i>Expression</i> ")"

Definitions

If $S1 \subseteq T$ and $S2 \subseteq T$,

$$S1 - S2 = \{ x \mid x \in T \wedge (x \in S1 \wedge x \notin S2) \}$$

$$S1 \cup S2 = \{ x \mid x \in T \wedge (x \in S1 \vee x \in S2) \}$$

$$S1 \cap S2 = \{ x \mid x \in T \wedge (x \in S1 \wedge x \in S2) \}$$

If $U \in \mathbb{P}(\mathbb{P}(T))$,

$$\text{union}(U) = \{ x \mid x \in T \wedge \exists y. (y \in U \wedge x \in y) \}$$

If $U \in \mathbb{P}1(\mathbb{P}(T))$,

$$\text{inter}(U) = \{ x \mid x \in T \wedge \forall y. (y \in U \Rightarrow x \in y) \}$$

If $\forall x. (P \Rightarrow S \subseteq T)$,

$$\bigcup x.(P \mid S) = \{ y \mid y \in T \wedge \exists z. (z \in T \wedge P \wedge y \in S) \}$$

If $\forall x. (P \Rightarrow S \subseteq T)$ and $\exists x. (P)$,

$$\bigcap x.(P \mid S) = \{ y \mid y \in T \wedge \forall z. (z \in T \wedge P \Rightarrow y \in S) \}$$

Typing rules

In expressions $S1 - S2$, $S1 \cup S2$ and $S1 \cap S2$, sets $S1$ and $S2$ must be the same type in form $\mathbb{P}(T)$. The type of these expressions is $\mathbb{P}(T)$.

In expressions $\text{union}(EI)$ and $\text{inter}(EI)$, EI must be a set of sets, with type $\mathbb{P}(\mathbb{P}(T))$. The type of these expressions is $\mathbb{P}(T)$.

In expressions $\bigcup X. (P \mid S)$ and $\bigcap X. (P \mid S)$, X refers to a list of identifiers, P is a predicate that must type X and S is a type $\mathbb{P}(T)$ set. The type of these expressions is $\mathbb{P}(T)$.

Well defineness

Expression	Well defineness condition
$\text{inter}(E)$	E must not be empty
$\bigcap X.(P E)$	$\{X P\}$ must not be empty

Scope

In expressions $\bigcup X.(P | S)$ and $\bigcap X.(P | S)$, the range of the list of identifiers X is predicate P and expression S .

Restrictions

- The X variables introduced by the expressions matching $\bigcup X.(P | S)$ and $\bigcap X.(P | S)$ must be typed by an abstract data typing predicate (refer to section 3.1 *Typing foundations*), located in a list of conjunctions at the highest level of syntax analysis in P . These variables cannot be used in P before they are typed.

Description

Let E and F be sets.

- $E - F$, represents the difference between sets E and F , i.e. the set of elements that belong to E but not to F .
- $E \cup F$ represents the union between sets E and F , i.e. the set of elements that belong to E or to F .
- $E \cap F$ represents the intersection between sets E and F i.e. the set of elements that belong to E and to F .

Let ENS be a set of sets.

- $\text{union}(ENS)$ represents the generalized union of elements of ENS , i.e. the set obtained by the union of sets forming the elements of ENS .
- $\text{inter}(EI)$ represents the generalized intersection of elements of ENS , i.e. the set obtained in the intersection of sets that make up the elements of ENS .

Let X be a list of variables, P a predicate that types the list of X variables and then expresses a property on X . With E a set defined as a function of X .

- $\bigcup X.(P | E)$ represents the union of sets E indexed using a list of X variables that verify predicate P . If P is false, then the quantified union represents the empty set.
- $\bigcap X.(P | E)$ represents the intersection of sets E indexed using a list of X variables that verify predicate P . If P is false, then the quantified intersection is meaningless.

Examples

With $EI = \{-1, 0, 3, 7, 8\}$ and $FI = \{-3, -1, 4, 7, 9\}$,

$$EI - FI = \{0, 3, 8\}$$

$$EI \cup FI = \{-3, -1, 0, 3, 4, 7, 8, 9\}$$

$$EI \cap FI = \{-1, 7\}$$

With $SI = \{\{1\}, \{1, 2\}, \{1, 3\}\}$,

$$\text{union}(SI) = \{1, 2, 3\}$$

$$\text{inter}(SI) = \{1\}$$

With $E2 = \{2, 4\}$,

$$\bigcup_{xI}.(xI \in E2 \mid \{yI \mid yI \in \mathbb{N} \wedge yI \leq xI\}) = \{0, 1, 2\} \cup \{0, 1, 2, 3, 4\} = \{0, 1, 2, 3, 4\}$$

$$\bigcap_{xI}.(xI \in E2 \mid \{yI \mid yI \in \mathbb{N} \wedge yI \leq xI\}) = \{0, 1, 2\} \cap \{0, 1, 2, 3, 4\} = \{0, 1, 2\}$$

5.9 Record expressions

Operator

struct	Set of records
rec	record in extension
'	access to a record field (quote operator)

Syntax

$Set_of_records$	$::=$	"struct" "(" (Ident ":" Expression) ⁺ ")"
$Extensive_record$	$::=$	"rec" "(" ([Ident ":"] Expression) ⁺ ")"
$Record_field$	$::=$	Expression "'" Ident

Typing rules

Let n be an integer greater than or equal to 1, and i be an integer included between 1 and n .

In the expression $struct(Ident1 : E1, \dots, Identn : En)$, Ei must be of type $\mathbb{P}(Ti)$. Then the type of the expression is $\mathbb{P}(struct(Ident1 : T1, \dots, Identn : Tn))$. In the expression $rec(Ident1 : x1, \dots, Identn : xn)$, let Ti be the type of xi . Then the type of the expression is $struct(Ident1 : T1, \dots, Identn : Tn)$. In the expression $rec(x1, \dots, xn)$, let Ti be the type of each xi expression. Then the type of the expression matches $struct(Ident1 : T1, \dots, Identn : Tn)$, where the $Ident_i$ are identifiers which are distinct two by two. In the expression $Record'Ident_i$, the type of $Record$ must match $struct(Ident1 : T1, \dots, Identn : Tn)$, where $Ident_i$ is the i th label of the record type. Then, the type of the expression is Ti .

Restrictions

1. In the expression $struct(Ident1 : E1, \dots, Identn : En)$, the names of the $Ident_i$ fields must be distinct two by two.
2. In the expression $rec(Ident1 : x1, \dots, Identn : xn)$, the names of the $Ident_i$ fields must be distinct two by two.
3. A record in extension, without labels, of the form $rec(x1, \dots, xn)$, cannot be used to type a piece of data.

Description

Let n be an integer greater than or equal to 1, and i be an integer included between 1 and n .

- Let $E1, \dots, En$ be sets and $Ident1, \dots, Identn$ be identifiers which are distinct two by two, then $struct(Ident1 : E1, \dots, Identn : En)$ refers to a set of record data. This set is the non-empty ordered collection of the n sets $E1, \dots, En$ called fields of the set of records. Each field has a name $Ident_i$ called its label.
- Let $x1, \dots, xn$ be expressions and $Ident1, \dots, Identn$ be identifiers distinct two by two, then $rec(Ident1 : x1, \dots, Identn : xn)$ refers to a record data, in which the value of each $Ident_i$ fields is xi . If this record data is not used to type another piece of data (refer to Typing of abstract data), then the labels are optional. The simplified notation $rec(x1, \dots, xn)$ can then be used instead of the previous one.
- Let rc be a piece of record data of which one of the labels is $identi$, then the

expression $rc'identi$ built up with the help of the *quote* operator refers to the value of the *identi* field of the record data *rc*.

Examples

$RES_SET = \text{struct}(\text{Mark} : 0..20, \text{Good_enough} : \text{BOOL})$ represents a set of records with two fields. The first field is called *Mark* and it refers to the set 0..20. The second field is called *Good_enough* and refers to the Boolean set.

$\text{Result} = \text{rec}(\text{Mark} : 12, \text{Good_enough} : \text{TRUE})$ represents a piece of data belonging to the *RES_SET* set. The value of the *Mark* field is 12 and that of the *Good_enough* field is TRUE. If the *result* data has already been typed, then the preceding writing can be simplified to $\text{Result} = (12, \text{TRUE})$.

$\text{Result}'\text{Mark}$ represents the value of the field *Mark*, that is to say 12.

5.10 Sets of Relations

Operator

\leftrightarrow Setsof relations

Syntax

Relations::= *Expression* " \leftrightarrow " *Expression*

Definition

$$X \leftrightarrow Y = \mathbb{P}(X \times Y)$$

Typing rule

In the expression $X \leftrightarrow Y$, X must be a $\mathbb{P}(T1)$ type and Y must be a $\mathbb{P}(T2)$ type. The type of $X \leftrightarrow Y$ is $\mathbb{P}(\mathbb{P}(T1 \times T2))$.

Description

Let E and F be sets. A relation of E into F is a set of couples $(x \mapsto y)$, where x is an element of E and where y is an element of F .

$E \leftrightarrow F$ designates the set of relations between set E into set F . This is another notation for $\mathbb{P}(E \times F)$.

Examples

$0..5 \leftrightarrow \text{BOOL}$ represents the set of relations for the interval $0..5$ in the BOOL set. The following relations belong to this set:

$$rel1 = \{(0 \mapsto \text{FALSE}), (1 \mapsto \text{TRUE}), (2 \mapsto \text{FALSE}), (3 \mapsto \text{TRUE}), (4 \mapsto \text{FALSE}), (5 \mapsto \text{TRUE})\}$$

$$rel2 = \{(0 \mapsto \text{FALSE}), (0 \mapsto \text{TRUE}), (3 \mapsto \text{TRUE})\}$$

$$rel3 = \emptyset$$

5.11 Expressions of Relations

Operator

id	Identity
r^{-1}	Reverse
prj ₁	First projection
prj ₂	Second projection
;	Composition
\otimes	Direct product
	Parallel product

Syntax

<i>Identity</i>	::=	"id" "(" <i>Expression</i> ")"
<i>Inverse</i>	::=	<i>Expression</i> ⁻¹
<i>First_projection</i>	::=	"prj ₁ " "(" <i>Expression</i> "," <i>Expression</i> ")"
<i>Second_projection</i>	::=	"prj ₂ " "(" <i>Expression</i> "," <i>Expression</i> ")"
<i>Composition</i>	::=	<i>Expression</i> ";" <i>Expression</i>
<i>Direct_product</i>	::=	<i>Expression</i> " \otimes " <i>Expression</i>
<i>Parallel_product</i>	::=	<i>Expression</i> " " <i>Expression</i>

Definitions

$$\text{id}(X) = \{ x, y \mid x \in X \wedge y = x \}$$

If $R \in X \leftrightarrow Y$,

$$R^{-1} = \{ y, x \mid (y \mapsto x) \in R \wedge (x \mapsto y) \in R \}$$

$$\text{prj}_1(E, F) = \{ x, y, z \mid x, y, z \in E \times F \times E \wedge z = x \}$$

$$\text{prj}_2(E, F) = \{ x, y, z \mid x, y, z \in E \times F \times F \wedge z = y \}$$

If $R_1 \in T \leftrightarrow U$ and $R_2 \in U \leftrightarrow V$,

$$R_1 ; R_2 = \{ x, z \mid x, z \in T \times V \wedge \exists y. (y \in U \wedge (x \mapsto y) \in R_1 \wedge (y \mapsto z) \in R_2) \}$$

If $R_1 \in T \leftrightarrow U$ and $R_2 \in T \leftrightarrow V$,

$$R_1 \otimes R_2 = \{ x, (y, z) \mid x, (y, z) \in T \times (U \times V) \wedge (x \mapsto y) \in R_1 \wedge (x \mapsto z) \in R_2 \}$$

If $R_1 \in T \leftrightarrow U$ and $R_2 \in V \leftrightarrow W$,

$$R_1 \parallel R_2 = \{ (x, y), (z, a) \mid (x, y), (z, a) \in (T \times V) \times (U \times W) \wedge (x \mapsto z) \in R_1 \wedge (y \mapsto a) \in R_2 \}$$

Typing rule

In the expression $\text{id}(E)$, E must be a $\mathbb{P}(T)$ type. Then $\text{id}(E)$ is a $\mathbb{P}(T \times T)$ type.

In the expression R^{-1} , R must be a $\mathbb{P}(T \times U)$ type. Then R^{-1} is a $\mathbb{P}(U \times T)$ type relation.

In the expressions $\text{prj}_1(E, F)$ and $\text{prj}_2(E, F)$, E and F must be $\mathbb{P}(T)$ and $\mathbb{P}(U)$ types. Then $\text{prj}_1(E, F)$ is a type $\mathbb{P}(T \times U \times T)$ relation and $\text{prj}_2(E, F)$ is a type $\mathbb{P}(T \times U \times U)$ relation.

In the expression $E ; F$, E must be a $\mathbb{P}(T \times U)$ type and F must be a $\mathbb{P}(U \times V)$ type. Then $E ; F$ is a $\mathbb{P}(T \times V)$ type relation.

In the expression $E \otimes F$, E must be a $\mathbb{P}(T \times U)$ type and F must be a $\mathbb{P}(T \times V)$ type. Then $E \otimes F$ is a $\mathbb{P}(T \times (U \times V))$ type relation.

In the expression $E \parallel F$, E must be a $\mathbb{P}(T \times U)$ type and F must be a $\mathbb{P}(V \times W)$ type. Then, $E \parallel F$ is a $\mathbb{P}((T \times V) \times (U \times W))$ type relation.

Restriction

The operators $;$ and \parallel representing respectively the composition and the parallel product of two relations, must not be used if there is a possible ambiguity with the operators representing substitutions. Brackets must be use in order to avoid these ambiguities. For example, it is forbidden to write $R3:=R1 ; R2$. Use $R3:=(R1 ; R2)$ instead.

Description

- Let X be a set, $\text{id}(X)$ represents the identity relation of X in itself, i.e. the relation that associates any element of X with this same element.
- Let R be a relation, R^{-1} represents the inverse relation of R . i.e. the relation made up of inverse couples to those in R . If $(x \mapsto y) \in R$ then $(y \mapsto x) \in R^{-1}$.

Let X and Y be sets,

- $\text{prj}_1(X, Y)$ represents the first projection relation of $X \times Y$ in X , that with any couple $(x1, y1)$ of $X \times Y$ associates with the first component $x1$ of the couple.
- $\text{prj}_2(X, Y)$ represents the second projection relation of $X \times Y$ in Y , that with any couple $(x1 \mapsto y1)$ of $X \times Y$ associates with the second component $y1$ of the couple.
- Let $R1$ be a relation from set A to set B and $R2$ a relation from set B to set C . Then $R1 ; R2$ represents the composition of $R1$ and $R2$. It contains the set of couples $(a1 \mapsto c1)$ so that there exists an element $b1$ in B so that $(a1 \mapsto b1) \in R1$ and $(b1 \mapsto c1) \in R2$.
- Let $R1$ be a relation from set A to set B and $R2$ a relation from set A to set C . Then $R1 \otimes R2$ represents the direct product of $R1$ and $R2$. This relation contains the set of couples $a1 \mapsto (b1 \mapsto c1)$ so that there exists a couple $(a1 \mapsto b1)$ from $R1$ and a couple $(a1 \mapsto c1)$ from $R2$.
- Let $R1$ be a relation from set A to set B and $R2$ a relation from set C to set D . Then $R1 \parallel R2$ represents the parallel product of $R1$ and $R2$. This relation contains the set of couples $((a1 \mapsto c1) \mapsto (b1 \mapsto d1))$ so that there exists a couple $(a1 \mapsto b1)$ de $R1$ and a couple $(c1 \mapsto d1)$ of $R2$.

Examples

With $E1 = \{3, 5\}$,

$$\text{id}(E1) = \{(3 \mapsto 3), (5 \mapsto 5)\}$$

With $R1 = \{(0 \mapsto 4), (2 \mapsto 4), (2 \mapsto 7), (3 \mapsto 3)\}$,

$$R1^{-1} = \{(4 \mapsto 0), (4 \mapsto 2), (7 \mapsto 2), (3 \mapsto 3)\}$$

With $E1 = \{0, 1\}$ and $F1 = \{-1, 2\}$,

$$\text{prj}_1(E1, F1) = \{((0 \mapsto -1) \mapsto 0), ((0 \mapsto 2) \mapsto 0), ((1 \mapsto -1) \mapsto 1), ((1 \mapsto 2) \mapsto 1)\}$$

$$\text{prj}_2(E1, F1) = \{((0 \mapsto -1) \mapsto -1), ((0 \mapsto 2) \mapsto 2), ((1 \mapsto -1) \mapsto -1), ((1 \mapsto 2) \mapsto 2)\}$$

With $R1 = \{(0 \mapsto 2), (1 \mapsto 5), (2 \mapsto 5), (3 \mapsto 7)\}$ and $R2 = \{(0 \mapsto 0), (2 \mapsto -1), (5 \mapsto 8), (6 \mapsto 9)\}$,

$$R1 ; R2 = \{(0 \mapsto -1), (1 \mapsto 8), (2 \mapsto 8)\}$$

With $R1 = \{(0 \mapsto 0), (1 \mapsto 10), (2 \mapsto 20)\}$ and $R2 = \{(0 \mapsto 0), (1 \mapsto 20), (2 \mapsto 40), (3 \mapsto 60)\}$,

$$R1 \otimes R2 = \{ (0 \mapsto (0 \mapsto 0)), \\ (1 \mapsto (10 \mapsto 20)), \\ (2 \mapsto (20 \mapsto 40)) \}$$

With $R1 = \{(0 \mapsto 7), (1 \mapsto 6)\}$ and $R2 = \{(10 \mapsto 11), (12 \mapsto 12)\}$,

$$R1 \parallel R2 = \{ ((0 \mapsto 10) \mapsto (7 \mapsto 11)), \\ ((0 \mapsto 12) \mapsto (7 \mapsto 12)), \\ ((1 \mapsto 10) \mapsto (6 \mapsto 11)), \\ ((1 \mapsto 12) \mapsto (6 \mapsto 12)) \}$$

5.12 Expressions of Relations (continued)

Operator

R^n	Iteration
R^*	Transitive and reflexive closure
R^+	Transitive closure

Syntax

<i>Iteration</i>	::=	<i>Expression</i> ^{<i>Expression</i>}
<i>Reflexive_closure</i>	::=	<i>Expression</i> [*]
<i>Closure</i>	::=	<i>Expression</i> ⁺

Definitions

Let R be a relation of a set E in itself and n a natural integer.

$$R^1 = iR$$

$$R^{n+1} = R ; R^n$$

$$R^* = \bigcup_{n \in \mathbb{N}} (n \mid R^n)$$

$$R^+ = \bigcup_{n \in \mathbb{N}_1} (n \mid R^n)$$

Typing rules

In the expression R^n , R is a $\mathbb{P}(t \times t)$ type and n is a \mathbb{Z} type. The type of expression is $\mathbb{P}(t \times t)$.

In the expressions R^* and R^+ , R must be a $\mathbb{P}(t \times t)$ type. The type of expressions is $\mathbb{P}(t \times t)$.

Well defineness

Expression	Well defineness condition
R^n	n must belong to \mathbb{N}

Description

Let R be a relation of a set E in itself and n a natural integer.

- R^n represents the relation R iterated n times in relation to the composition operator.
- R^* represents the transitive and reflexive closure of R . It is the smallest relation that contains R that is transitive and reflexive.
- R^+ represents the transitive closure of R . It is the smallest relation containing R that is transitive.

Examples

With $E = \{1, 2, 3\}$, $R = \{(1 \mapsto 3), (2 \mapsto 1), (2 \mapsto 2), (3 \mapsto 3)\}$,

$$R^1 = R$$

$$R^2 = \{(1 \mapsto 3), (2 \mapsto 1), (2 \mapsto 2), (2 \mapsto 3), (3 \mapsto 3)\}$$

$$R^+ = R^2$$

$$R^* = \{(1 \mapsto 1), (1 \mapsto 3), (2 \mapsto 1), (2 \mapsto 2), (2 \mapsto 3), (3 \mapsto 3)\}$$

5.13 Expressions of Relations (continued)

Operator

dom	Domain
ran	Range
[]	Image

Syntax

Domain	::=	"dom" "(" <i>Expression</i> ")"
Range	::=	"ran" "(" <i>Expression</i> ")"
Image	::=	<i>Expression</i> "[" <i>Expression</i> "]"

Definitions

If $R \in E1 \leftrightarrow E2$,

$$\text{dom} = \{ x \mid x \in E1 \wedge \exists y . (y \in E2 \wedge (x \mapsto y) \in R) \}$$

$$\text{ran} = \{ y \mid y \in E2 \wedge \exists x . (x \in E1 \wedge (x \mapsto y) \in R) \}$$

If $R \in E1 \leftrightarrow E2$ and $F \subseteq E1$,

$$R[F] = \{ y \mid y \in E2 \wedge \exists x . (x \in F \wedge (x \mapsto y) \in R) \}$$

Typing rule

In the expressions $\text{dom}(R)$ and $\text{ran}(R)$, R must be a $\mathbb{P}(T \times V)$ type relation. The type of dom will be $\mathbb{P}(T)$ and the type of ran is $\mathbb{P}(V)$.

In the expression $R[E]$, R must be a $\mathbb{P}(T \times V)$ type relation and E must be a $\mathbb{P}(T)$ type set. Then the expression is a $\mathbb{P}(V)$ type one.

Description

Let R be the relation of set A with set B .

- dom designates the domain of R , i.e. the set of elements a in A for which there exists an element b in B such as $(a \mapsto b) \in R$.
- ran designates the range of R (*range*), i.e. the set of elements b in B for which there exists an element $(a \mapsto b)$ in R .

Let E be a part of A ,

- $R[E]$ designates the image of E for R . This is the set of elements of B that are associated with an element of E via the relation R .

Examples

With $R = \{(0 \mapsto 4), (2 \mapsto 4), (2 \mapsto 7), (3 \mapsto 3)\}$,

$$\text{dom} = \{0, 2, 3\}$$

$$\text{ran} = \{4, 7, 3\}$$

With $E = \{-1, 0, 1, 2\}$,

$$R[E] = \{4, 7\}$$

5.14 Expressions of Relations (continued)

Operator

\triangleleft	Restriction in the domain
$\triangleleft\!\!\!\!<$	Subtraction in the domain
\triangleright	Restriction in the range
$\triangleright\!\!\!\!$	Subtraction in the range
$\triangleleft\!\!\!\!$	Overwrite

Syntax

<i>Domain_restriction</i>	::=	<i>Expression</i> " \triangleleft " <i>Expression</i>
<i>Domain_subtraction</i>	::=	<i>Expression</i> " $\triangleleft\!\!\!\!$ " <i>Expression</i>
<i>Range_restriction</i>	::=	<i>Expression</i> " \triangleright " <i>Expression</i>
<i>Subtractions_range</i>	::=	<i>Expression</i> " $\triangleright\!\!\!\!$ " <i>Expression</i>
<i>Overwrite</i>	::=	<i>Expression</i> " $\triangleleft\!\!\!\!$ " <i>Expression</i>

Definitions

If $R \in E1 \leftrightarrow E2$ and $F \subseteq E1$,

$$F \triangleleft R = \{ x, y \mid (x \mapsto y) \in R \wedge x \in F \}$$

$$F \triangleleft\!\!\!\! R = \{ x, y \mid (x \mapsto y) \in R \wedge x \notin F \}$$

If $R \in E1 \leftrightarrow E2$ and $F \subseteq E2$,

$$R \triangleright F = \{ x, y \mid (x \mapsto y) \in R \wedge y \in F \}$$

$$R \triangleright\!\!\!\! F = \{ x, y \mid (x \mapsto y) \in R \wedge y \notin F \}$$

If $R \in E1 \leftrightarrow E2$ and $Q \in E1 \leftrightarrow E2$,

$$Q \triangleleft\!\!\!\! R = \{ x, y \mid (x, y) \in E1 \times E2 \wedge ((x \mapsto y) \in Q \wedge x \notin \text{dom}(R)) \vee (x \mapsto y) \in R \}$$

Typing rule

In the expressions $E1 \triangleleft R$ and $E1 \triangleleft\!\!\!\! R$, R must be a type $\mathbb{P}(T \times V)$ relation and $E1$ must be a $\mathbb{P}(T)$ set. The type of expressions is $\mathbb{P}(T \times V)$.

In the expressions $R \triangleright F$ and $R \triangleright\!\!\!\! F$, R must be a type $\mathbb{P}(T \times V)$ relation and F must be a type $\mathbb{P}(V)$ set. The type of expression is $\mathbb{P}(T \times V)$.

In the expression $R1 \triangleleft\!\!\!\! R2$, $R1$ and $R2$ must be type $\mathbb{P}(T \times V)$ relations. The expression type is $\mathbb{P}(T \times V)$.

Description

Let $R1$ and $R2$ be relations, $E1$ and $F1$ be sets.

- $E1 \triangleleft R1$ designates the restriction on the domain of $R1$ for set $E1$. This is the set of couples $(x1 \mapsto y1)$ of $R1$ for which $x1$ belongs to $E1$.
- $E1 \triangleleft\!\!\!\! R1$ designates the subtraction on the domain of $R1$ for set $E1$. This is the set of couples $(x1 \mapsto y1)$ of $R1$ for which $x1$ does not belong to $E1$.
- $R1 \triangleright F1$ designates the restriction on the range of $R1$ for set $F1$. This is the set of couples $(x1 \mapsto y1)$ of $R1$ for which $y1$ belongs to $F1$.

- $R1 \triangleright F1$ designates the subtraction on the range of $R1$ for set $F1$. This is the set of couples $(x1 \mapsto y1)$ of $R1$ for which $y1$ does not belong to $F1$.
- $R1 \triangleleft R2$ designates the overwriting of $R1$ with $R2$. This is the relation comprising elements of $R2$ and of $R1$ the first element of which does not belong to the domain of $R2$. Therefore in the relation obtained, the elements of $R2$ in $(x1 \mapsto zi)$ notation overwrite any elements $(x1 \mapsto yi)$ in $R1$.

Examples

With relation $R = \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$, with sets $E = \{1, 2, 3\}$ and $F = \{5, 7, 9\}$,

$$E \triangleleft R = \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 9)\}$$

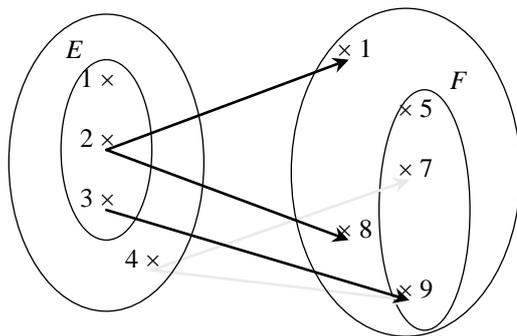
$$E \triangleleft R = \{(4 \mapsto 7), (4 \mapsto 9)\}$$

$$R \triangleright F = \{(3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$$

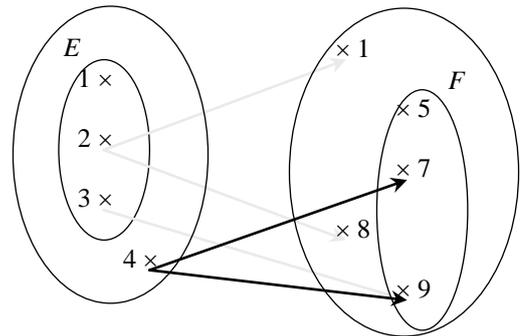
$$R \triangleright F = \{(2 \mapsto 1), (2 \mapsto 8)\}$$

With relations $R1 = \{(2 \mapsto 1), (2 \mapsto 8), (3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$ and $R2 = \{(0 \mapsto -1), (1 \mapsto 7), (2 \mapsto 9)\}$,

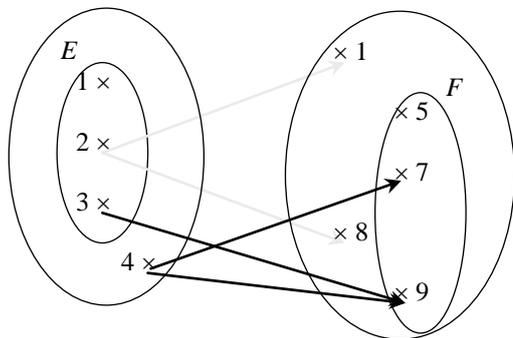
$$R1 \triangleleft R2 = \{(0 \mapsto -1), (1 \mapsto 7), (2 \mapsto 9), (3 \mapsto 9), (4 \mapsto 7), (4 \mapsto 9)\}$$



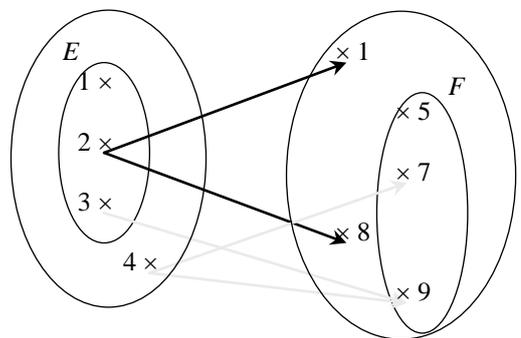
$E \triangleleft R$



$E \triangleleft R$



$R \triangleright F$



$R \triangleright F$

5.15 Sets of Functions

Operator

\rightarrow	Partial functions
\rightarrow	Total functions
\rightsquigarrow	Partial injections
\rightsquigarrow	Total injections
\twoheadrightarrow	Partial surjections
\twoheadrightarrow	Total surjections
\twoheadrightarrow	Total bijections

Syntax

<i>Partial_functions</i>	::=	<i>Expression</i> " \rightarrow " <i>Expression</i>
<i>Total_functions</i>	::=	<i>Expression</i> " \rightarrow " <i>Expression</i>
<i>Partial_injections</i>	::=	<i>Expression</i> " \rightsquigarrow " <i>Expression</i>
<i>Total_injections</i>	::=	<i>Expression</i> " \rightsquigarrow " <i>Expression</i>
<i>Partial_surjections</i>	::=	<i>Expression</i> " \twoheadrightarrow " <i>Expression</i>
<i>Total_surjections</i>	::=	<i>Expression</i> " \twoheadrightarrow " <i>Expression</i>
<i>Total_bijections</i>	::=	<i>Expression</i> " \twoheadrightarrow " <i>Expression</i>

Definitions

$$\begin{aligned}
 E1 \rightarrow E2 &= \{ r \mid r \in E1 \leftrightarrow E2 \wedge (r^{-1} ; r) \subseteq \text{id}(E2) \} \\
 E1 \rightarrow E2 &= \{ f \mid f \in E1 \rightarrow E2 \wedge \text{dom}(f) = E1 \} \\
 E1 \rightsquigarrow E2 &= \{ f \mid f \in E1 \rightarrow E2 \wedge f^1 \in E2 \rightarrow E1 \} \\
 E1 \rightsquigarrow E2 &= E1 \rightsquigarrow E2 \cap E1 \rightarrow E2 \\
 E1 \twoheadrightarrow E2 &= \{ f \mid f \in E1 \rightarrow E2 \wedge \text{ran}(f) = E2 \} \\
 E1 \twoheadrightarrow E2 &= E1 \twoheadrightarrow E2 \cap E1 \rightarrow E2 \\
 E1 \twoheadrightarrow E2 &= E1 \rightsquigarrow E2 \cap E1 \twoheadrightarrow E2
 \end{aligned}$$

Typing rule

In expressions $E1 \rightarrow E2$, $E1 \rightarrow E2$, $E1 \rightsquigarrow E2$, $E1 \rightsquigarrow E2$, $E1 \twoheadrightarrow E2$, $E1 \twoheadrightarrow E2$, $E1 \rightsquigarrow E2$, $E1 \twoheadrightarrow E2$, $E1$ and $E2$ are any types $\mathbb{P}(T1)$ and $\mathbb{P}(T2)$. The type of expressions is $\mathbb{P}(\mathbb{P}(T1 \times T2))$.

Description

With $E1$ and $F1$ in sets.

- $E1 \rightarrow E2$ designated the set of partial functions of $E1$ in $E2$. A partial function of $E1$ in $E2$ is a relation that does not contain two distinct couples with the same first element.
- $E1 \rightarrow E2$ designates the set of total functions of $E1$ in $E2$. A total function of $E1$ in $E2$ is a partial function whose domain is exactly $E1$ (and is not only included in $E1$ as is the case for a partial function).
- $E1 \rightsquigarrow E2$ designated the set of partial injections of $E1$ in $E2$. A partial injection of $E1$ in $E2$ is a partial function that with two different elements of $E1$ associates two different elements of $E2$ using a partial function. The reverse of a partial injection

of $E1$ in $E2$ is a partial function of $E2$ in $E1$. The total injection concept, the symbol of which is \hookrightarrow , is defined in a similar way.

- $E1 \twoheadrightarrow E2$ designate the set of partial surjections of $E1$ in $E2$. A partial surjection of $E1$ in $E2$ is a partial function that for any element of $E2$ has a correspondence with an element of $E1$. The concept of a total surjection, the symbol of which is \twoheadrightarrow , is defined in a similar way.

Examples

If $r1 = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 2)\}$,

then $r1 \in \{0, 1, 2, 3\} \mapsto \{0, 1, 2\}$

and $r1 \in \{0, 1, 2\} \rightarrow \{0, 1, 2\}$

If $r2 = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 3)\}$,

then $r2 \in \{0, 1, 2, 3\} \twoheadrightarrow \{0, 1, 2, 3\}$

and $r2 \in \{0, 1, 2\} \twoheadrightarrow \{0, 1, 2, 3\}$

If $r3 = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 2)\}$,

then $r3 \in \{0, 1, 2, 3\} \twoheadrightarrow \{1, 2\}$

and $r3 \in \{0, 1, 2\} \twoheadrightarrow \{1, 2\}$

If $r4 = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 3)\}$,

then $r4 \in \{0, 1, 2\} \twoheadrightarrow \{1, 2, 3\}$

5.16 Expressions of Functions

Operator

λ	Lambda-expression
$f()$	Evaluation of the function
fnc	Transformed into a function
rel	Transformed into a relation

Syntax

<i>Lambda_expression</i>	::=	" λ " <i>List_ident</i> "." "(" <i>Predicate</i> " " <i>Expression</i> ")"
<i>Evaluation_functions</i>	::=	<i>Expression</i> "(" <i>Expression</i> ")"
<i>Transformed_function</i>	::=	"fnc" "(" <i>Expression</i> ")"
<i>Transformed_relation</i>	::=	"rel" "(" <i>Expression</i> ")"

Definitions

If $\forall x . (x \in t \Rightarrow E \in u)$,

$$\lambda x . (x \in t \wedge P | E) = \{ x, y | x, y \in t \times u \wedge P \wedge y = E \} \quad \text{where } y \text{ is not free in } x, t, P \text{ and } E$$

If $f \in t \mapsto u$ and $E \in \text{dom}(f)$,

$$f(E) = \text{choice} (f[\{E\}])$$

Remember, the choice operator (refer to [B-Book] section 2.1.2) applied to a non empty set, designates a “privileged” element of this set. In this case, the set in question, $f[\{E\}]$, has only one element. The privileged element in this set can therefore only be this element. Important, the choice operator must not be confused with the “limited selection” substitution (refer to 6.6) that uses the keyword CHOICE.

Typing rule

In expression $\lambda(X).(P|E)$, X designates a list of identifiers that are distinct two by two, P is a predicate that must start by typing X , X is then a Cartesian product type $T1 \times \dots \times Tn$. E is a type T expression. The expression is a $\mathbb{P}(T1 \times \dots \times Tn \times T)$ type.

In expression $f1(y1)$, $f1$ is a $\mathbb{P}(T1 \times T2)$ type expression and $y1$ must be a $T1$ type. The expression is a $T2$ type.

In expression $\text{fnc}(R)$, R must be a $\mathbb{P}(T1 \times T2)$ type relation. The expression type is $\mathbb{P}(T1 \times \mathbb{P}(T2))$.

In expression $\text{rel}(R)$, R represents a relation, the elements of which are sets, its type must be $\mathbb{P}(T1 \times \mathbb{P}(T2))$. The expression type is $\mathbb{P}(T1 \times T2)$.

Well defineness

Expression	Well defineness condition
$f(x)$	$x \in \text{dom}(f) \quad \wedge \quad f \in \text{dom}(f) \rightarrow \text{ran}(f)$
$\text{rel}(f)$	$f \in \text{dom}(f) \rightarrow \text{ran}(f)$

Scope

In expressions $\lambda xI.(P|E)$ and $\lambda(X).(P|E)$, the scope of xI and X is predicate P and expression E .

Restriction

- The variables X introduced by the expressions matching $\lambda X.(P|E)$ must be typed by an abstract data typing predicate (refer to section 3.1 *Typing foundations*), located in a list of conjunctions at the highest syntax analysis level of P . These variables cannot be used in P before they have been typed.

Description

- A lambda expression is used to define a function by giving, in an expression, its value for each element of the function domain. Let xI be an identifier and $P1$ a predicate that starts by typing xI and $E1$ an expression that depends on xI . Then $\lambda xI.(P1|E1)$ designates a lambda expression. This is the function made up of couples $(xI, E1)$ for each element of xI that verifies $P1$. In the same way, if X is a list of identifiers that are distinct when taken two by two (xI, \dots, xn) , then $\lambda(X).(P1|E1)$ is the function made up of elements $(xI, \dots, xn, E1)$ where X verifies $P1$.
- Let RI be a function of $E1$ in fI and let xI be an element of $E1$. Then $E1(xI)$ designates the only yI element of fI so that the couple $(xI \mapsto yI)$ belongs to RI . The expression has a meaning, only if xI belongs to the domain of RI .
- With RI a relation of $E1$ in fI . Then $fnc(RI)$ designates the transforming into a function of RI . This is the function of $E1$ in $\mathbb{P}(fI)$ that for each element xI of the domain of RI assigns the set of elements of fI linked to xI via the relation RI .
- Let Fct be a function of $E1$ in $\mathbb{P}(fI)$. Then $rel(Fct)$ designates the transforming into a relation of Fct . This is the relation of $E1$ in fI made up of couples $(xI \mapsto yI)$ so that xI belongs to the domain of Fct and so that yI belongs to the element associated with xI by the relation Fct .

Examples

The lambda expression: $\lambda xI.(xI \in \mathbb{Z} | xI \times 2)$ defines the multiply by 2 function in \mathbb{Z} .

With function $fI = \{(0 \mapsto 6), (1 \mapsto 2), (3 \mapsto 6), (4 \mapsto -5)\}$,

$$fI(3) = 6$$

With relation $RI = \{(0 \mapsto 1), (0 \mapsto 2), (1 \mapsto 1), (1 \mapsto 7), (2 \mapsto 3)\}$,

$$fnc(RI) = \{(0 \mapsto \{1, 2\}), (1 \mapsto \{1, 7\}), (2 \mapsto \{3\})\}$$

With function $fI = \{(-1 \mapsto \{0, 2\}), (1 \mapsto \{6, 8\}), (3 \mapsto \{3\})\}$,

$$rel(fI) = \{(-1 \mapsto 0), (-1 \mapsto 2), (1 \mapsto 6), (1 \mapsto 8), (3 \mapsto 3)\}$$

5.17 Sets of Sequences

Operator

seq	Sequences
seq ₁	Non empty sequences
iseq	Injective sequences
iseq ₁	Injective non empty sequences
perm	Permutations
[]	Empty sequence
[...]	Sequence in extension

Syntax

<i>Sequences</i>	::=	"seq" "(" <i>Expression</i> ")"
<i>Sequences_non_empty</i>	::=	"seq ₁ " "(" <i>Expression</i> ")"
<i>Sequences_injective</i>	::=	"iseq" "(" <i>Expression</i> ")"
<i>Sequences_inj_non_empty</i>	::=	"iseq ₁ " "(" <i>Expression</i> ")"
<i>Permutations</i>	::=	"perm" "(" <i>Expression</i> ")"
<i>Empty_sequence</i>	::=	"[]"
<i>Sequence_extension</i>	::=	"[" <i>Expression</i> ⁺ "]"

Typing rule

In the seq(E), seq₁(E), iseq(E), iseq₁(E), perm(E) expressions, E must designate a $\mathbb{P}(t)$ type set. Then expressions are $\mathbb{P}(\mathbb{Z} \times t)$ type.

The empty sequence [] has no established fixed type. It may take the type of any sequence. Its type is therefore matching $\mathbb{P}(\mathbb{Z} \times t)$.

In the extension sequence [E_1, \dots, E_n], the elements E_1, \dots, E_n of the sequence must all be of the same type t . The sequence type is then $\mathbb{P}(\mathbb{Z} \times t)$.

Well defineness

Expression	Well defineness condition
perm(E)	E must be a finite set

Definitions

$$\begin{aligned} \text{seq}(E) &= \bigcup n . (n \in \mathbb{N} \mid 1..n \rightarrow E) \\ \text{seq}_1(E) &= \text{seq}(E) - \{\emptyset\} \\ \text{iseq}(E) &= \{s \mid s \in \text{seq}(E) \wedge s \in \mathbb{N}_1 \twoheadrightarrow E\} \\ \text{iseq}_1(E) &= \text{iseq}(E) - \{\emptyset\} \\ \text{perm}(E) &= \{s \mid s \in \text{iseq}(E) \wedge s \in \mathbb{N}_1 \twoheadrightarrow E\} \\ [] &= \emptyset \\ [E_1, \dots, E_n] &= \{(1 \mapsto E_1), \dots, (n \mapsto E_n)\} \end{aligned}$$

Description

The sequences handled in B language are finite sequences. The sequences are modeled as total functions of an integer interval matching $1..n$, where $n \in \mathbb{N}$, to any set E . As the sequences are functions, all of the function handling operators apply to the sequences. It is said that the n^{th} element in a sequence is the value e_n so that $(n \mapsto e_n)$ belongs to the sequence.

- $\text{seq}(E)$ designates the set of sequences, the elements of which belong to set E .
- $\text{seq}_1(E)$ designates the set of sequences in set E and which is not the empty sequence.
- $\text{iseq}(E)$ designates the set of injective sequences in set E .
- $\text{iseq}_1(E)$ designates the set of injective sequences in set E and which is not the empty sequence.
- $\text{perm}(E)$ designates the set of bijective sequences in set E . These sequences are called permutations. Note that set E must be finite.
- $[]$ designates the empty sequence empty. It is a function that does not have any elements. The empty sequence is none other than the empty set \emptyset .
- $[E_1, \dots, E_n]$ designates the extension sequence, the n elements of which are in sequence E_1, \dots, E_n .

Examples

With set $E = \{0, 1, 2\}$,

$[] \in \text{seq}(E)$, $[0] \in \text{seq}(E)$, $[1, 2, 0] \in \text{seq}(E)$, $[0, 2, 2, 0, 1, 0, 0] \in \text{seq}(E)$

$[0] \in \text{seq}_1(E)$, $[1, 2, 0] \in \text{seq}_1(E)$, $[0, 2, 2, 0, 1, 0, 0] \in \text{seq}_1(E)$

$[] \in \text{iseq}(E)$, $[1] \in \text{iseq}(E)$, $[1, 2, 0] \in \text{iseq}(E)$, $[0, 2] \in \text{iseq}(E)$, but $[0, 1, 0] \notin \text{iseq}(E)$

$[1] \in \text{iseq}_1(E)$, $[1, 2, 0] \in \text{iseq}_1(E)$, $[0, 2] \in \text{iseq}_1(E)$, but $[0, 1, 0] \notin \text{iseq}_1(E)$

$[0, 1, 2] \in \text{perm}(E)$, $[1, 0, 2] \in \text{perm}(E)$, $[2, 1, 0] \in \text{perm}(E)$, but $[0, 1] \notin \text{perm}(E)$

5.18 Sequence Expressions

Operator

size	Size
first	First element
last	Last element
front	Front
tail	Tail
rev	Reverse

Syntax

<i>Sequence_size</i>	::=	"size" "(" <i>Expression</i> ")"
<i>Sequence_first_element</i>	::=	"first" "(" <i>Expression</i> ")"
<i>Sequence_last_element</i>	::=	"last" "(" <i>Expression</i> ")"
<i>Sequence_front</i>	::=	"front" "(" <i>Expression</i> ")"
<i>Sequence_tail</i>	::=	"tail" "(" <i>Expression</i> ")"
<i>Reverse_sequence</i>	::=	"rev" "(" <i>Expression</i> ")"

Typing rule

In expressions $\text{size}(E)$, $\text{first}(E)$, $\text{last}(E)$, $\text{front}(E)$, $\text{tail}(E)$ and $\text{rev}(E)$, E must designate a $\mathbb{P}(\mathbb{Z} \times T)$ type sequence. The type of $\text{size}(E)$ is an integer type. The type of expressions $\text{first}(E)$ and $\text{last}(E)$ is t . The type of sequences $\text{front}(E)$, $\text{tail}(E)$ and $\text{rev}(E)$ is $\mathbb{P}(\mathbb{Z} \times T)$.

Well defineness

Expression	Well defineness condition
$\text{size}(s)$	$s \in \text{seq}(\text{ran}(s))$
$\text{first}(s)$	
$\text{last}(s)$	
$\text{front}(s)$	
$\text{tail}(s)$	
$\text{rev}(s)$	

Definitions

$\text{size}(s) = \text{card}(s)$
$\text{first}(s) = s(1)$
$\text{last}(s) = s(\text{size}(s))$
$\text{front}(s) = s \uparrow (\text{size}(s) - 1)$
$\text{tail}(s) = s \downarrow 1$
$\text{rev}(s) = \lambda i . (i \in 1.. \text{size}(s) \mid s(\text{size}(s) - i + 1))$

Description

Let $s1$ be a sequence and $s2$ a non empty sequence,

- `size (s1)` represents the number of elements in the sequence,
- `first (s2)` represents the first element in $s2$,
- `last (s2)` represents the last element in $s2$,
- `front (s2)` represents sequence $s2$, without its last element,
- `tail (s2)` represents sequence $s2$, without its first element,
- `rev (s1)` represents the sequence comprising the same elements as $s1$, but in reverse order.

Examples

With sequence $s1 = [5, 7, -2, 1]$,

`size (s1) = 4`

`first (s1) = 5`

`last (s1) = 1`

`front (s1) = [5, 7, -2]`

`tail (s1) = [7, -2, 1]`

`rev (s1) = [1, -2, 7, 5]`

5.19 Sequence Expressions (continued)

Operator

$\hat{\ }^{\circ}$	Concatenation
\rightarrow	Insert in front
\leftarrow	Insert at tail
\uparrow	Restrict in front
\downarrow	Restrict at tail
conc	General concatenation

Syntax

<i>Concatenation</i>	::=	<i>Expression</i> " $\hat{\ }^{\circ}$ " <i>Expression</i>
<i>Insert_front</i>	::=	<i>Expression</i> " \rightarrow " <i>Expression</i>
<i>Insert_tail</i>	::=	<i>Expression</i> " \leftarrow " <i>Expression</i>
<i>Restrict_front</i>	::=	<i>Expression</i> " \uparrow " <i>Expression</i>
<i>Restrict_tail</i>	::=	<i>Expression</i> " \downarrow " <i>Expression</i>
<i>Concat_general</i>	::=	"conc" "(" <i>Expression</i> ")"

Typing rule

In expressions $s1 \hat{\ }^{\circ} s2$, $s1$ and $s2$ designate $\mathbb{P}(\mathbb{Z} \times T)$ type sequences.

In expressions $El \rightarrow s1$ and $s1 \leftarrow El$, $s1$ refers to a $\mathbb{P}(\mathbb{Z} \times T)$ type sequence and El refers to an element in the sequence, of type T .

In expressions $s1 \uparrow n$ and $s1 \downarrow n$, $s1$ refers to a type $\mathbb{P}(\mathbb{Z} \times T)$ sequence and n must be an integer type.

In expression $\text{conc}(s1)$, $s1$ refers to a sequence, the elements of which are sequences of the same type. $s1$ must therefore be a $\mathbb{P}(\mathbb{Z} \times \mathbb{P}(\mathbb{Z} \times T))$ type.

Well defineness

Expression	Well defineness condition
$s \uparrow n$	
$s \downarrow n$	n must belong to the interval $0 .. \text{size}(s)$

Definitions

$s \uparrow n = (1 .. n) \triangleleft s$
$s \downarrow n = \lambda i. (i \in 1 .. \text{size}(s) - n \mid s(n+i))$
$s1 \hat{\ }^{\circ} s2 \triangleq s1 \cup \lambda i. (i \in \text{size}(s1)+1 .. \text{size}(s1)+\text{size}(s2) \mid s2(i - \text{size}(s1)))$
$x \rightarrow s \triangleq \{1 \mapsto x\} \cup \lambda i. (i \in 2 .. \text{size}(s)+1 \mid s(i - 1))$
$s \leftarrow x \triangleq s \cup \{\text{size}(s) + 1 \mapsto x\}$
$\text{conc}([]) \triangleq []$
$\text{conc}(x \rightarrow s) \triangleq x \hat{\ }^{\circ} \text{conc}(s)$

Description

With $s1$ and $s2$ as sequences,

- $s1 \hat{\ } s2$ represents the sequence obtained by concatenating in sequence, sequences $s1$ and $s2$.

With $s1$ a sequence, $E1$ a new element and n a relative integer,

- $E1 \rightarrow s1$ represents the sequence obtained by inserting at the front of the sequence $s1$ the new element $E1$.
- $s1 \leftarrow E1$ represents the sequence obtained by inserting at the tail of the sequence $s1$ the new element $E1$.
- $s1 \uparrow n$ represents the sequence obtained by restricting $s1$ to the n first elements. If n is greater than the size of $s1$, the sequence obtained is $s1$.
- If n is less than or equal to the size of $s1$ then, $s1 \downarrow n$ represents the sequence obtained by eliminating from $s1$ its n first elements.

Let $S1$ be a sequence whose elements are sequences,

- $\text{conc}(S1)$ represents the sequence obtained by concatenating in sequence all of the sequences that are elements of $S1$.

Examples

With sequences $s1 = [3, 1]$ and $s2 = [0, -2, 4]$,

$$s1 \hat{\ } s2 = [3, 1, 0, -2, 4]$$

$$2 \rightarrow s1 = [2, 3, 1]$$

$$s1 \leftarrow 2 = [3, 1, 2]$$

$$s2 \uparrow 2 = [0, -2], s2 \uparrow 4 = [0, -2, 4]$$

$$s2 \downarrow 2 = [4], s2 \downarrow 3 = [], s2 \downarrow 0 = [0, -2, 4]$$

With sequence $S1 = [[2, 5], [-1, -2, 9], [], [5]]$,

$$\text{conc}(S1) = [2, 5, -1, -2, 9, 5]$$

5.20 Tree sets

Operator

tree	Trees
btree	Binary Trees

Syntax

Trees	::=	“tree” (“ Expression “)
Binary_trees	::=	“btree” (“ Expression “)

Typing Rules

In the expressions $\text{tree}(S)$ and $\text{btree}(S)$, S must designate a set of the type $\mathbb{P}(T)$. The type of $\text{tree}(S)$ and of $\text{btree}(S)$ is $\mathbb{P}(\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T))$.

Definitions

ins	$\hat{=} \lambda i . (i \in \mathbb{N} \mid \lambda i . (i \in \text{seq}(\mathbb{N}) \mid i \rightarrow S))$
cns	$\hat{=} \lambda t . (t \in \text{seq}(\mathbb{F}(\text{seq}(\mathbb{N}_1))) \mid \{[]\} \cup \bigcup i . (t \in \text{dom}(t) \mid \text{ins}(i)[t(i)]))$
T	$\hat{=} \text{cns}[\text{seq}(T)]$
$\text{tree}(S)$	$\hat{=} \bigcup t . (t \in T \mid t \rightarrow S)$
$\text{btree}(S)$	$\hat{=} \{ t \mid t \in \text{tree}(S) \wedge \forall n . (n \in \text{dom}(t) \Rightarrow \text{arity}(t, n) \in \{0, 2\}) \}$

Description

Trees that are modelised in B language are finite and decorated trees which have ordered branches.

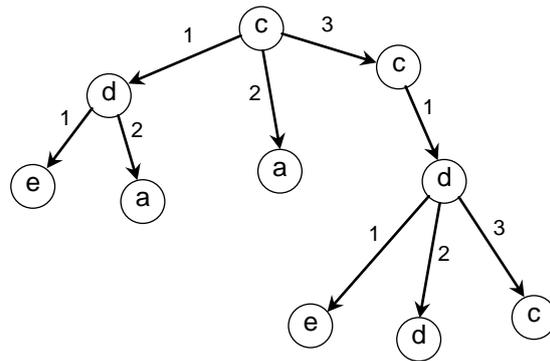
- A tree is composed of a finite set of nodes. If a branch links the A node to the B node, then it is said that A is the father of B and that B is the son of A. A tree node can have no more than one father. The only node that does not have a father is called the root of the tree. A tree being never empty always has a root. A node can have from 0 to n sons. This number is called the arity of the node. A node which as a null arity is called a leaf. If a node has got one or several sons, then their order has a signification, the branches leading to the sons are numbered from 1 to n.
- A node is represented mathematically by the sequence of the branch numbers linking the root of the tree to the node. The root of the tree is represented by an empty sequence.
- The tree is said to be decorated because an element of a given set S is associated with each of its nodes.
- $\text{tree}(S)$ is the set of trees decorated with elements of the S set. An element of this set is a total function of a finite set of nodes towards an S set. Each node is represented by the sequence of branches leading to the node from the root of the tree. A node branch is identified by an element of \mathbb{N}_1 .
- $\text{btree}(S)$ is the set of binary trees decorated with elements of the S set. A binary tree is a tree for which the arity of each node is equal to 0 or to 2. A node of a binary tree is therefore either a leaf, or a node with two branches which are called right branch and left branch.

Example

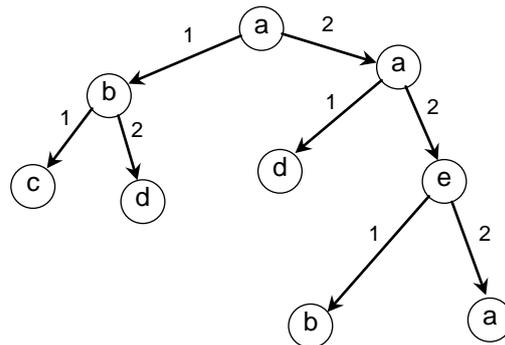
The graphic representations of the trees given below use the following convention. A circle which contains the value associated with the node represents a node. Arrows represent the branches from the father to the son, numbered from 1 to n.

Let S be the following enumerated set : $S = \{ a, b, c, d, e \}$

The A tree is an element of the tree(S) :

$$A = \{ [] \mapsto c, \\ [1] \mapsto d, \\ [1, 1] \mapsto e, \\ [1, 2] \mapsto a, \\ [2] \mapsto a, \\ [3] \mapsto c, \\ [3, 1] \mapsto d, \\ [3, 1, 1] \mapsto e, \\ [3, 1, 2] \mapsto d, \\ [3, 1, 3] \mapsto c \}$$


The B tree is an element of betree(S) :

$$B = \{ [] \mapsto a, \\ [1] \mapsto b, \\ [1, 1] \mapsto c, \\ [1, 2] \mapsto d, \\ [2] \mapsto a, \\ [2, 1] \mapsto d, \\ [2, 2] \mapsto e, \\ [2, 2, 1] \mapsto b, \\ [2, 2, 2] \mapsto a \}$$


5.21 Tree Expressions

Operator

const	constructor
top	root
sons	sons
prefix	prefixed flattening
postfix	postfixed flattening
sizet	size
mirror	symmetry

Syntax

<i>Tree_constructor</i>	::=	"const" "(" <i>Expression</i> "," <i>Expression</i> ")"
<i>Tree_root</i>	::=	"top" "(" <i>Expression</i> ")"
<i>Tree_sons</i>	::=	"sons" "(" <i>Expression</i> ")"
<i>Prefixed_flattening</i>	::=	"prefix" "(" <i>Expression</i> ")"
<i>Postfixed_flattening</i>	::=	"postfix" "(" <i>Expression</i> ")"
<i>Tree_size</i>	::=	"sizet" "(" <i>Expression</i> ")"
<i>Tree_symmetry</i>	::=	"mirror" "(" <i>Expression</i> ")"

Typing Rules

In the expressions $\text{const}(x,q)$, $\text{top}(t)$, $\text{sons}(t)$, $\text{prefix}(t)$, $\text{postfix}(t)$, $\text{sizet}(t)$, $\text{mirror}(t)$, t must be a tree of type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$, x must be of type T and q must be a sequence of trees of type $\mathbb{P}(\mathbb{Z} \times \mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T))$. The type of the expressions $\text{const}(x,q)$ and $\text{mirror}(t)$ is $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$. The type of $\text{top}(t)$ is T . The type of $\text{sons}(t)$ is $\mathbb{P}(\mathbb{Z} \times \mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T))$. The type of the expressions $\text{prefix}(t,n)$ and $\text{postfix}(t,n,i)$ is $\mathbb{P}(\mathbb{Z} \times T)$.

Restriction

1. In the expression $\text{const}(x)$, x must be a couple formed by two elements separated by a comma.

Proper Definition

Expression	Well defineness condition
$\text{const}(x, q)$	$x \in S \wedge q \in \text{seq}(\text{tree}(S))$
$\text{top}(t)$	$t \in \text{tree}(S)$
$\text{sons}(t)$	
$\text{prefix}(t)$	
$\text{postfix}(t)$	
$\text{sizet}(t)$	
$\text{mirror}(t)$	

Definitions

$$\text{const}(x, q) \cong \{ [] \mapsto x \} \cup \bigcup i . (i \in \text{dom}(q) \mid \text{ins}(i)^{-1} ; q(i))$$

$$\text{top} \cong \text{const}^{-1} ; \text{prj}_1(S, \text{seq}(\text{tree}(S)))$$

sons	$\hat{=}$ $\text{const}^{-1} ; \text{prj}_2 (S, \text{seq} (\text{tree} (S)))$
prefix (t)	$\hat{=}$ $\text{prefix} (t) \rightarrow \text{conc} (\text{sons} (t) ; \text{prefix})$
postfix (t)	$\hat{=}$ $\text{conc} (\text{sons} (t) ; \text{postfix}) \leftarrow \text{top} (t)$
sizet (t)	$\hat{=}$ $\text{succ} (\text{sum} (\text{sons} (t) ; \text{sizet}))$
mirror (t)	$\hat{=}$ $\text{const} (\text{top} (t), \text{rev} (\text{sons} (t) ; \text{mirror}))$

Description

Let S be a set, let t be a tree decorated with elements of S , let x be an element of S and let T an element of $\text{seq} (\text{seq} (\mathbb{N}_1) \rightarrow S)$,

- $\text{const}(x,q)$ represents the tree whose root is associated with x and whose sons are the elements of the sequence q .
- $\text{top} (t)$ represents the value associated to the root of the t tree,
- $\text{sons}(t)$ represents the sequence of sons of the root of the t tree.
- $\text{prefix}(t)$ represents the prefixed flattening of the elements of S borne by the t tree, in a sequence. This sequence can be defined in a recursive way. If t is a leaf, $\text{prefix}(t)$ is the sequence containing the value associated with the leaf node. Otherwise $\text{prefix}(t)$ can be obtained by concatenating the sequence containing the value associated with the root of t , and the prefixed sequences of each son of the root of t , taken in order.
- $\text{postfix}(t)$ represents the postfix flattening of the elements of S borne by the t tree, in a sequence. This sequence can be defined in a recursive way. If t is a leaf, $\text{postfix}(t)$ is the sequence containing the value associated with the leaf node. Otherwise $\text{postfix}(t)$ can be obtained by and the postfix sequences of each son of the root of t , taken in order, and concatenating the sequence containing the value associated with the root of t .
- $\text{sizet}(t)$ represents the size of the t tree. It is the number of nodes of t . This expression can be defined in a recursive way,
- $\text{mirror} (t)$ represents the tree which is symmetrical to t . It is the tree which is built from t by reverting for each node the order of the sons of the node.

Examples

Let S be the following enumerated set : $S = \{ a, b, c, d, e \}$

Let $A1, A2, A3$ be trees bearing elements of S :

$$A1 = \{ [] \mapsto d, [1] \mapsto e, [2] \mapsto a \}$$

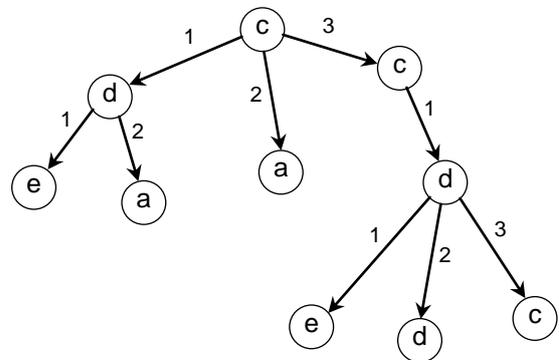
$$A2 = \{ [] \mapsto a \}$$

$$A3 = \{ [] \mapsto c, [1] \mapsto d, [1, 1] \mapsto e, \\ [1, 2] \mapsto d, [1, 3] \mapsto c \}$$

$$A = \text{const} (c, [A1, A2, A3])$$

Then :

$$A = \{ [] \mapsto c, \\ [1] \mapsto d, [1, 1] \mapsto e, [1, 2] \mapsto a, \\ [2] \mapsto a, \\ [3] \mapsto c, [3, 1] \mapsto d, [3, 1, 1] \mapsto e, [3, 1, 2] \mapsto d, [3, 1, 3] \mapsto c \}$$



$$\text{top}(A) = c$$

$$\text{sons}(A) = [A1, A2, A3]$$

$$\text{prefix}(A) = [c, d, e, a, a, c, d, e, d, c]$$

$$\text{postfix}(A) = [e, a, d, a, e, d, c, d, c, c]$$

$$\text{size}(A) = 10$$

$$\text{mirror}(A) = \{ \begin{array}{l} [] \mapsto c, \\ [1] \mapsto c, [1, 1] \mapsto d, [1, 1, 1] \mapsto c, [1, 1, 2] \mapsto d, [1, 1, 3] \mapsto e, \\ [2] \mapsto a, \\ [3] \mapsto d, [3, 1] \mapsto a, [3, 2] \mapsto e \end{array} \}$$

5.22 Tree nodes expressions

Operator

rank	rank of a node
father	father of a node
son	son of a node
subtree	subtree
arity	arity of a node

Syntax

<i>Node_rank</i>	::=	"rank" "(" <i>Expression</i> "," <i>Expression</i> ")"
<i>Node_father</i>	::=	"father" "(" <i>Expression</i> "," <i>Expression</i> ")"
<i>Node_son</i>	::=	"son" "(" <i>Expression</i> "," <i>Expression</i> "," <i>Expression</i> ")"
<i>Node_subtree</i>	::=	"subtree" "(" <i>Expression</i> "," <i>Expression</i> ")"
<i>Node_arity</i>	::=	"arity" "(" <i>Expression</i> "," <i>Expression</i> ")"

Typing rules

In the expressions $\text{rank}(t,n)$, $\text{father}(t,n)$, $\text{son}(t,n,i)$, $\text{subtree}(t,n)$, $\text{arity}(t,n)$, t must be a tree of type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$, n must be a sequence of type $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$ and i must be of type \mathbb{Z} . The type of the expressions $\text{rank}(t,n)$ and $\text{arity}(t,n)$ is the integer type \mathbb{Z} . The type of the expressions $\text{father}(t,n)$, $\text{son}(t,n,i)$ is $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$. The type of $\text{subtree}(t,n)$ is $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$

Restrictions

1. In the expressions $\text{rank}(x)$, $\text{father}(x)$, $\text{subtree}(x)$, and $\text{arity}(x)$, x must be a couple made up of two elements separated by a comma.
2. In the expression $\text{son}(x)$, x must be a list of three elements separated by commas.

Well defineness

Expression	Well defineness condition
$\text{rank}(t, n)$	$t \in \text{tree}(S) \wedge n \in \text{dom}(t) - \{ [] \}$
$\text{father}(t, n)$	$t \in \text{tree}(S) \wedge n \in \text{dom}(t) - \{ [] \}$
$\text{son}(t, n, i)$	$t \in \text{tree}(S) \wedge n \leftarrow i \in \text{dom}(t)$
$\text{subtree}(t, n)$	$t \in \text{tree}(S) \wedge n \in \text{dom}(t)$
$\text{arity}(t, n)$	$t \in \text{tree}(S) \wedge n \in \text{dom}(t)$

Definitions

$\text{rank}(t, n)$	$\hat{=}$	$\text{last}(n)$
$\text{father}(t, n)$	$\hat{=}$	$\text{front}(n)$
$\text{son}(t, n, i)$	$\hat{=}$	$n \leftarrow i$
$\text{subtree}(t, n)$	$\hat{=}$	$\text{cat}(n) ; t$
$\text{arity}(t, n)$	$\hat{=}$	$\text{size}(\text{sons}(\text{subtree}(t, n)))$

Description

Let S be a set, let t be a tree decorated with elements of S , let n and m be the nodes of t , (in the form of a sequence of branch numbers describing the path to the node from the root). The node m must be different from the root of the tree. Finally, let i be one of the branches starting at n .

- $\text{rank}(t, m)$ represents the rank of the branch linking the father of m to m .
- $\text{father}(t, n)$ represents the father of the node n in the tree t ,
- $\text{son}(t, n, i)$ represents the son of rank i of the node n of the tree t ,
- $\text{subtree}(t, n)$ represents the subtree of the tree t whose root is the node n .
- $\text{arity}(t, n)$ represents the arity of the node n in the tree t , that is to say, the number of sons of n .

Examples

Let S be the following enumerated set : $S = \{ a, b, c, d, e \}$

Let A be the following tree bearing items of S :

$$A = \{ [] \mapsto c, \\ [1] \mapsto d, [1, 1] \mapsto e, [1, 2] \mapsto a, \\ [2] \mapsto a, \\ [3] \mapsto c, [3, 1] \mapsto d, [3, 1, 1] \mapsto e, \\ [3, 1, 2] \mapsto d, [3, 1, 3] \mapsto c \}$$

Then

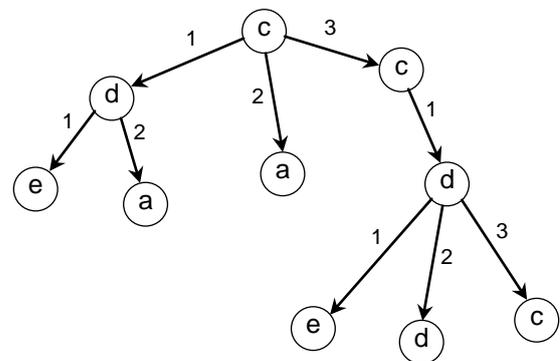
$$\text{rank}(A, [3, 1, 2]) = 2$$

$$\text{father}(A, [3, 1, 2]) = [3, 1]$$

$$\text{son}(A, [3, 1], 2) = [3, 1, 2]$$

$$\text{subtree}(A, [3, 1]) = \{ [] \mapsto d, [1] \mapsto e, [2] \mapsto d, [3] \mapsto c \}$$

$$\text{arity}(A, [1]) = 2$$



5.23 Binary Tree expressions

Operator

bin	Binary tree in extension
left	Left subtree
right	Right subtree
infix	Infix flattening

Syntax

<i>Extensive_binary_tree</i>	::=	“bin” “(“ <i>Expression</i> [“,” <i>Expression</i> ”,” <i>Expression</i>] ”)”
<i>Left_subtree</i>	::=	“left” “(“ <i>Expression</i> “)”
<i>Right_subtree</i>	::=	“right” “(“ <i>Expression</i> “)”
<i>Infix_flattening</i>	::=	“infix” “(“ <i>Expression</i> “)”

Typing Rules

In the expressions $\text{bin}(x)$, $\text{bin}(l,x,r)$, $\text{left}(t)$, $\text{right}(t)$, $\text{infix}(t)$, x is of type T , l and r must be trees of type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$, t must be a tree of type $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$. The type of the expressions $\text{bin}(x)$, $\text{bin}(l, x, r)$, $\text{left}(t)$, and $\text{right}(t)$ is $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times \mathbb{Z}) \times T)$. The type of $\text{infix}(t)$ is $\mathbb{P}(\mathbb{Z} \times T)$

Restrictions

1. In the expressions $\text{rank}(x)$, $\text{father}(x)$, $\text{subtree}(x)$ and $\text{arity}(x)$, x must be a couple formed by two elements separated by a comma.

Well defineness

Expression	Well defineness condition
$\text{bin}(x)$	$x : S$
$\text{bin}(l, x, r)$	$x : S \wedge l \in \text{btree}(S) \wedge r \in \text{btree}(S)$
$\text{left}(t)$	$t \in \text{btree}(S) \wedge \text{sons}(t) \neq []$
$\text{right}(t)$	$t \in \text{btree}(S) \wedge \text{sons}(t) \neq []$
$\text{infix}(t)$	$t \in \text{btree}(S)$

Definitions

$\text{bin}(x)$	$\hat{=} \text{const}(x, [])$
$\text{bin}(g, x, d)$	$\hat{=} \text{const}(x, [g, d])$
$\text{left}(t)$	$\hat{=} \text{first}(\text{sons}(t))$
$\text{right}(t)$	$\hat{=} \text{last}(\text{sons}(t))$
$\text{infix}(\text{bin}(x))$	$\hat{=} [x]$
$\text{infix}(\text{bin}(g, x, d))$	$\hat{=} \text{infix}(g) \wedge [x] \wedge \text{infix}(d)$

Description

Since binary trees are trees (refer to section 5.20 *Tree Sets*), all the tree expressions and the expressions of tree nodes from the preceding sections can be applied to binary trees. The expressions described below are specific to binary trees.

Let S be a set, let r be an element of S , let l, r, t and u be binary trees decorated with elements of S , and where t is not a leaf.

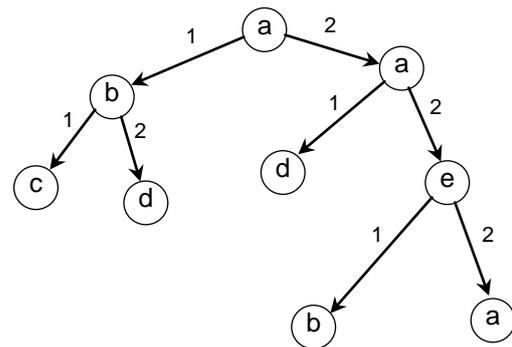
- $\text{bin}(x)$ represents the binary tree composed of a single node bearing the value x
- $\text{bin}(l,x,r)$ represents the binary tree whose root bears the value x , and whose root's left and right sons are the trees l and r ,
- $\text{left}(t)$ represents the left sub-tree of the t tree.
- $\text{right}(t)$ represents the right sub-tree of the t tree.
- $\text{infix}(u)$ represents the infix flattening in a sequence of the elements of S borne by the t tree. This sequence can be defined in a recursive way. If t is a leaf, $\text{infix}(t)$ is the sequence containing the value associated to the leaf node. Otherwise $\text{infix}(t)$ can be obtained by concatenating the infix sequence of the left sub-tree of t , to the sequence containing the value associated with the root of t , and to the infix sequence of the right sub-tree of t .

Examples

Let S be the following enumerated set : $S = \{ a, b, c, d, e \}$

Let B be the following tree bearing items of S :

- $B = \{ [] \mapsto a,$
 $[1] \mapsto b,$
 $[1, 1] \mapsto c,$
 $[1, 2] \mapsto d,$
 $[2] \mapsto a,$
 $[2, 1] \mapsto d,$
 $[2, 2] \mapsto e,$
 $[2, 2, 1] \mapsto b,$
 $[2, 2, 2] \mapsto a \}$



$$B = \text{bin} (\text{bin} (\text{bin} (c), b, \text{bin} (d)),$$

$$a,$$

$$\text{bin} (\text{bin} (d), a, \text{bin} (\text{bin} (b), e, \text{bin} (a))))$$

$$\text{left} (B) = \{ [] \mapsto b, [1] \mapsto c, [2] \mapsto d \}$$

$$\text{right} (B) = \{ [] \mapsto a, [1] \mapsto d, [2] \mapsto e, [2, 1] \mapsto b, [2, 2] \mapsto a \}$$

$$\text{infix} (B) = [c, b, d, a, d, a, b, e, a]$$

6. SUBSTITUTIONS

Syntax

The syntax of generalized substitutions is defined below:

```

Substitution ::=
    Block_substitution
    | Identity_substitution
    | Becomes_equal_substitution
    | Precondition_substitution
    | Assertion_substitution
    | Substitution_limited_choice
    | If_substitution
    | Select_substitution
    | Case_substitution
    | Any_substitution
    | Let_substitution
    | Becomes_elt_substitution
    | Becomes_such_that_substitution
    | Var_substitution
    | Substitution_call
    | Simultaneous_substitution
    | While_substitution
  
```

Description

The substitutions are mathematical notations defined as predicate transformers.

Let S be a substitution and P a predicate. Then, the notation:

$[S]P$ (read as “substitution S establishes predicate P ”)

represents the predicate obtained after transformation of P by substitution S . The following vocabulary is also used to designate this transformation: reference is made to the establishment by substitution S of postcondition P . Reference is also made to the application of substitution S to P . The substitutions allow modeling the dynamic aspect of B modules: their initialization as well as operations, as they allow establishing how the properties of module data are transformed by these operations.

Example

Here is a simple form of the “becomes equal” substitution. Let x and y be integer variables, then:

$[x := 3](y + x < 0)$

refers to the predicate obtained after replacement in predicate $y + x < 0$ of all free occurrences of variable x by the expression 3. The following predicate will then be obtained:

$y + 3 < 0$

Therefore the application of this “becomes equal” substitution does in fact correspond to the application of a substitution in that the value of x in the predicate $y + x < 0$ is replaced by 3.

Generalized substitutions

The generalized substitution language describes the set of substitutions that may be used in B language. Each generalized substitution is defined by specifying what the predicate obtained is after application of the substitution to a given predicate.

In the following chapters, a detailed description is provided of the generalized substitutions. Here is the list:

BEGIN	block substitution
skip	identity substitution
:=	becomes equal to substitution
:()	becomes such that substitution
:∈	becomes element of substitution
PRE	precondition substitution
ASSERT	assertion substitution
CHOICE	bounded choice substitution
IF	IF Conditional Substitution
SELECT	conditional bounded choice substitution
CASE	condition by case substitution
ANY	unbounded choice substitution
LET	local definition substitution
VAR	local variable substitution
;	sequencing substitution
WHILE	while loop substitution
←	operation call substitution
	simultaneously substitution

Using substitutions

Generalized substitutions are used in B language to describe the core of the initialization and of the operations on a component (refer to 7.22 *The INITIALISATION* and 7.23 *The OPERATIONS*). The mechanism for transforming a predicate with a substitution enables systematically generating the proof obligations relating to initialization and operations. For example, so that an abstract machine can be semantically correct, it is necessary to prove that each machine operation preserves the invariant. To do this, a proof obligation is generated with as assumption (among others) the machine invariant and as goal, the predicate obtained after transforming the invariant by the substitution that defines the operation. In the same way, proof obligations are generated to demonstrate that the initialization establishes the invariant and that the specification of an operation is preserved through refinement.

Properties of substitutions

Generalized substitutions have certain characteristics that are defined below:

Non determinism

A substitution has non deterministic behavior if it describes a number of possible behaviors without specifying which will in fact be chosen.

In B language, substitutions of machines and refinements may be non deterministic. The non determinism decreases with refinement. The implementation substitutions must be deterministic.

In the sections below, generalized substitutions are presented one after another. You are given for each substitution : its name, the type of components where it may be used, its syntax, and optionally its typing and scopes, its description and an example.

6.1 Block substitution

Syntax

Block_substitution ::= "BEGIN" *Substitution* "END"

Definition

Let S be a substitution and P a predicate, then :

$$\text{BEGIN } S \text{ END} \triangleq S$$

Description

The block substitution is used to bracket substitutions performed in sequence or in parallel.

Examples

```
BEGIN
  xI := xI + 1 ;
  yI := xI2
END
```

6.2 Identical substitution

Syntax

Identity_substitution ::= "skip"

Definition

Let P be a predicate, then:

[skip] $P \Leftrightarrow P$

Description

The identity (skip) substitution takes no action. It does not change the predicate that it is applied to. The identity substitution is especially useful for showing that some branches in an IF, CASE or SELECT substitution perform no action.

6.3 Becomes Equal Substitution

Syntax

$$\begin{aligned} \text{Becomes_equal_substitution} &::= \\ &\text{Ident}^{+,\dots} \text{" := " Expression}^{+,\dots} \\ &| \text{Ident} \text{" (" Expression}^{+,\dots} \text{") " := " Expression} \\ &| \text{Ident} \text{" (" Ident }^{+} \text{" := " Expression} \end{aligned}$$

Definitions

1. Let x be a variable, E an expression and P a predicate, then :

$$[x := E] P$$

is the predicate obtained by replacing all of the free occurrences of x in P with E .

2. Let x and y two variables, E and F expressions of the same type as x and y , and P a predicate. Let z an intermediate variables non-free in x, y, E, F and P . Then:

$$[x, y := E, F] P \Leftrightarrow [z := F][x := E][y := z] P$$

The construction of a multiple “becomes equal” substitution for a list of variables is then defined iteratively.

3. Let fct be a function, $E1$ and $E2$ expressions and P a predicate. The expression $E1$ must belong to the fct domain. Then:

$$[fct(E1) := E2] P \Leftrightarrow [fct := fct \leftarrow \{E1 \mapsto E2\}] P$$

is the predicate obtained by replacing all of the free occurrences of fct in R with fct overwritten by $\{E1 \mapsto E2\}$.

4. Let n be an integer > 0 and i an integer $\in 1..n$. Let rc be a record data with type struct $(Ident_1 : T_1, \dots, Ident_n : T_n)$, y an expression and P a predicate. Then:

$$[rc'Ident_i := y] P \Leftrightarrow [rc := \text{rec}(Ident_1 : rc'Ident_1, \dots, Ident_i : y, \dots, Ident_n : rc'Ident_n)] P$$

This definition can be extended to nested record fields. For example, let rc a data record with type struct $(c_1^1 : T_1^1, \dots, c_i^1 : \text{struct}(c_1^2 : T_1^2, \dots, c_j^2 : T_j^2, c_m^2 : T_m^2), \dots, c_n^1 : T_n^1)$, y an expression and P a predicate. Then:

$$\begin{aligned} [rc'c_i^1'c_j^2 := y] P \Leftrightarrow [rc := \text{rec}(c_1^1 : rc'c_1^1, \dots, \\ c_i^1 : \text{rec}(c_1^2 : rc'c_1^1'c_1^2, \dots, c_j^2 : y, \dots, c_m^2 : rc'c_1^1'c_m^2), \dots, \\ c_n^1 : rc'c_n^1)] P \end{aligned}$$

Typing rule

In the expression $x1 := E1$, $x1$ and $E1$ must be of the same type t .

In the expression $x1, \dots, xn := E1, \dots, Em$, $(x1, \dots, xn)$ and $E1, \dots, Em$ must be of the same type $t1 \times \dots \times tn$.

In the expression $f(x1, \dots, xn) := e$, f must be of the type $\mathbb{P}(T_1 \times \dots \times T_n \times T_0)$. Then, each x_i must be of the type T_i and e must be of the type T_0 .

In the expression $rc'Ident_i := y$, rc must be of the type struct $(Ident_1 : T_1, \dots, Ident_n : T_n)$. Then, y must be of the type T_i . This rule can be extended to embedded record fields. In the case of two embedded levels, this rule yields : in the substitution $rc'c_i^1'c_j^2 := y$, rc must be of the type struct $(c_1^1 : T_1^1, \dots, c_i^1 : \text{struct}(c_1^2 : T_1^2, \dots, c_j^2 : T_j^2, c_m^2 : T_m^2), \dots, c_n^1 : T_n^1)$. Then, y must be of the type T_j^2 .

Restrictions

1. In a “becomes equal” substitution on a list of variables, the variables must differ two by two.
2. In a “becomes equal” substitution on a list of variables, the number of expressions on the right hand side must be the same as the number of variables.
3. Each variable modified by a “becomes equal” substitution must be accessible in write mode.

Description

The “becomes equal” substitution is used to replace a variable with an expression. It is defined in a number of forms:

1. “becomes equal” substitution for a variable

Applying the $x := E$ substitution replaces the non-free occurrences of variable x with expression E .

2. “becomes equal” substitution for a list of variables

The multiple “becomes equal” substitution is then defined for a list of variables. This kind of multiple substitution corresponds in semantically to a list of single “becomes equal” substitutions performed in parallel.

3. “becomes equal” substitution for a function element

Then the “becomes equal” substitution is defined for a function element is an abbreviation to replace one element of the function with an expression. The $fcn(x) := y$ notation in fact refers to the “becomes equal” substitution of fcn by itself, overwritten for the x index level element by the value of expression y .

3. “becomes equal” substitution for a record

The “becomes equal” substitution for a record field is the abbreviation for the replacement of a record variable field by an expression.

Examples

```

x2 := x1 + 1 ;
tab1 := {(0 ↦ 3), (1 ↦ 1), (2 ↦ -7)} ;
tab1(1) := 12 ;
tab2 := tab3 ;
y1, y2, y3 := 0, 0, 0 ;
z1, z2 := z2, z1 ;
tab4(x1 + 2) := 1 ;
rc'c2 := FALSE ;
rdv'Date'Day := 13

```

6.4 Precondition Substitution

Syntax

Precondition_substitution ::= "PRE" *Predicate* "THEN" *Substitution* "END"

Definition

Let P and R be predicates and S a substitution.

$$[\text{PRE } P \text{ THEN } S \text{ END}] R \Leftrightarrow P \wedge [S] R$$

Description

Precondition substitution is used to set the preconditions required before calling an operation.

The proof obligation for preserving the invariant I of an operation defined by a precondition substitution $\text{PRE } P \text{ THEN } S \text{ END}$ is as follows:

$$I \wedge P \Rightarrow [S] I$$

When an operation with a $\text{PRE } P \text{ THEN } S \text{ END}$ precondition is called, the application of the precondition substitution corresponds to the precondition P and to the application of substitution S . If the precondition is not proved, then the substitution does not end. In other words, the performance described by a substitution with precondition is guaranteed only if the usage context of the use of the precondition is true.

It is necessary to distinguish the substitution precondition for the conditional substitution IF. The first is only usable if the predicate is valid, whereas the second is always performed, but its result depends on the validity of a predicate.

Example

```
PRE
  x1 ∈ NAT1
THEN
  x1 := x1 - 1
END
```

6.5 Assertion Substitution

Syntax

Assertion_substitution ::= "ASSERT" *Predicate* "THEN" *Substitution* "END"

Definition

Let P and R be predicates and S a substitution.

$$[\text{ASSERT } P \text{ THEN } S \text{ END}] R \Leftrightarrow P \wedge (P \Rightarrow [S] R)$$

Description

The assertion substitution `ASSERT P THEN S END` enables applying the substitution S under the assertion that the predicate P is true. This substitution is very close to the precondition substitution. Like for the precondition, if predicate P is not established, then the substitution fails. However, it is useful as it sets the assumption P for applying the substitution S , as well as for the substitutions that follow S until the end of the body of the operation or of the initialization in which it is used.

The role of the precondition and assertion substitutions differs. The primary purpose of a precondition is to set a type and to express properties relating to the input parameters of an operation, while an assertion substitution is used to provide assumptions for an operation that may ease the task of proving the operation.

The assertion substitution may be useful when proving a refinement of an operation containing conditional structures. If the operation and the refined operation both contain IF substitutions with equivalent conditions, then indicating this equivalence in an assertion substitution leads to immediate demonstration of the proof obligations of cross-branches refinement (cases where is considered the refinement of a substitution branch of the abstraction with condition P by a substitution branch of the refinement with a condition contradicting P). On the other hand, it is necessary to establish the assertion.

Examples

```
ASSERT
   $x1 < 5 \Leftrightarrow y2 = 0$ 
THEN
   $x1 := x1 - 5$ 
END
```

6.6 Bounded choice Substitution

Syntax

Substitution_limited_choice ::= "CHOICE" *Substitution* ("OR" *Substitution*) * "END"

Definition

Let S_1, \dots, S_n be substitutions (with $n \geq 2$) and P a predicate. Then the CHOICE substitution is defined by:

$$[\text{CHOICE } S_1 \text{ OR } \dots \text{ OR } S_n \text{ END}] P \Leftrightarrow [S_1] P \wedge \dots \wedge [S_n] P$$

Description

The bounded choice substitution enables defining a finite number of possible behaviors without specifying which will in fact be implemented. It therefore defines a non deterministic behavior.

Examples

```
CHOICE
  x1 := x1 + 1
OR
  x1 := x1 - 1
END
```

6.7 IF conditional substitution

Syntax

```

If_substitution ::=
    "IF" Predicate "THEN" Substitution
    [ "ELSIF" Predicate "THEN" Substitution ]*
    [ "ELSE" Substitution ]
    "END"

```

Definition

Let P_1, P_2, \dots, P_n and R be predicates (with $n \geq 1$) and let S_1, S_2, \dots, S_n and T be substitutions, then :

1. $[\text{IF } P_1 \text{ THEN } S_1 \text{ ELSE } T \text{ END}]R \Leftrightarrow (P_1 \wedge [S_1]R) \wedge (\neg P_1 \wedge [T]R)$
2. $\text{IF } P_1 \text{ THEN } S_1 \text{ END} = \text{IF } P \text{ THEN } S \text{ ELSE skip END}$
3. $[\text{IF } P_1 \text{ THEN } S_1 \text{ ELSIF } P_2 \text{ THEN } S_2 \dots \text{ ELSIF } P_n \text{ THEN } S_n \text{ ELSE } T \text{ END}]R \Leftrightarrow$
 $(P_1 \Rightarrow [S_1]R) \wedge ((\neg P_1 \wedge P_2) \Rightarrow [S_2]R) \wedge \dots \wedge ((\neg P_1 \wedge \dots \wedge \neg P_{n-1} \wedge P_n) \Rightarrow [S_n]R) \wedge$
 $((\neg P_1 \wedge \dots \wedge \neg P_n) \Rightarrow [T]R)$
4. $\text{IF } P_1 \text{ THEN } S_1 \text{ ELSIF } P_2 \text{ THEN } S_2 \dots \text{ ELSIF } P_n \text{ THEN } S_n \text{ END} =$
 $\text{IF } P_1 \text{ THEN } S_1 \text{ ELSIF } P_2 \text{ THEN } S_2 \dots \text{ ELSIF } P_n \text{ THEN } S_n \text{ ELSE skip END}$

Description

The conditional substitution IF is used to define, for a given program, a number of possible behaviors, depending on the validity of one or more predicates. The behavior defined by the conditional substitution IF is a deterministic one.

The IF conditional substitution is defined in various forms:

1. $\text{IF } P_1 \text{ THEN } S_1 \text{ ELSE } T \text{ END}$
 If predicate P_1 is true then the substitution S_1 applies, else substitution T applies.
2. $\text{IF } P_1 \text{ THEN } S_1 \text{ END}$
 The ELSE branch of an IF substitution is optional. If it is missing, by default it represents the identity substitution.
3. $\text{IF } P_1 \text{ THEN } S_1 \text{ ELSIF } P_2 \text{ THEN } S_2 \dots \text{ ELSIF } P_n \text{ THEN } S_n \text{ ELSE } T \text{ END}$
 The presence of an ELSIF branch in an IF substitution is equivalent to nesting another IF substitution in the ELSE branch of the first IF. It is possible to have any number of ELSIF branches in the same IF substitution.
4. $\text{IF } P_1 \text{ THEN } S_1 \text{ ELSIF } P_2 \text{ THEN } S_2 \dots \text{ ELSIF } P_n \text{ THEN } S_n \text{ END}$
 When an IF substitution has any number of ELSIF branches but no explicit ELSE branch, it is defined by default with an ELSE branch that contains the null substitution.

Examples

```

IF  $x1 \in \{ 2, 4, 8 \}$  THEN
     $x1 := x1 / 2$ 
END ;

```

```
IF  $y1 + z1 < 0$  THEN
   $y1 := -z1$ 
ELSE
   $y1 := 0$ 
END ;
```

```
IF  $x0 = 0$  THEN
   $sign := 0$ 
ELSIF  $x0 > 0$  THEN
   $sign := 1$ 
ELSE
   $sign := -1$ 
END
```

6.8 Conditional Bounded choice Substitution

Syntax

```

Select_substitution ::=
  "SELECT" Predicate "THEN" Substitution
  ( "WHEN" Predicate "THEN" Substitution ) *
  [ "ELSE" Substitution ]
  "END"

```

Definition

Let P_1, P_2, \dots, P_n and R be predicates, with $n \geq 1$. Let S_1, S_2, \dots, S_n and T be substitutions, then :

1. $[\text{SELECT } P_1 \text{ THEN } S_1 \text{ WHEN } P_2 \text{ THEN } S_2 \dots \text{ WHEN } P_n \text{ THEN } S_n \text{ END}] R \Leftrightarrow (P_1 \Rightarrow [S_1] R) \wedge (P_2 \Rightarrow [S_2] R) \wedge \dots \wedge (P_n \Rightarrow [S_n] R)$
2. $[\text{SELECT } P_1 \text{ THEN } S_1 \text{ WHEN } P_2 \text{ THEN } S_2 \dots \text{ WHEN } P_n \text{ THEN } S_n \text{ ELSE } T \text{ END}] R \Leftrightarrow (P_1 \Rightarrow [S_1] R) \wedge (P_2 \Rightarrow [S_2] R) \wedge \dots \wedge (P_n \Rightarrow [S_n] R) \wedge ((\neg P_1 \wedge \neg P_2 \wedge \dots \wedge \neg P_n) \Rightarrow [T] R)$
3. $[\text{SELECT } P_1 \text{ THEN } S_1 \text{ END}] R \Leftrightarrow P_1 \Rightarrow [S_1] R$
4. $[\text{SELECT } P_1 \text{ THEN } S_1 \text{ ELSE } T \text{ END}] R \Leftrightarrow P_1 \Rightarrow [S_1] R \wedge (\neg P_1 \Rightarrow [T] R)$

Description

The SELECT substitution is used to define for a given program, various possible behaviors depending on the validity of predicates. Each branch in the SELECT substitution describes one of these cases. It comprises a predicate and a substitution. If the predicate is true, then the substitution may apply. If all of the predicates are false and the SELECT substitution ends with an ELSE branch, then the substitution of the ELSE branch applies.

If the predicates for different branches are not mutually exclusive, a number of behaviors are possible and there is no specification of the one that will in fact be implemented. In this case the behavior of the SELECT substitution is non deterministic. In addition, if none of the predicates is valid and if the ELSE branch does not exist, then the substitution is not feasible.

Example

```

SELECT
  x ≥ 0 THEN
    y := x2
WHEN
  x ≤ 0 THEN
    y := -x2
END

```

6.9 Case Conditional Substitution

Syntax

```

Case_substitution ::=
    "CASE" Expression "OF"
    "EITHER" Simple_term+; "THEN" Substitution
    ( "OR" Simple_term+; "THEN" Substitution )+
    [ "ELSE" Substitution ]
    "END"
    "END"

```

Definitions

Let E be an expression, $L1, L2, \dots, Ln$ lists of literal constants, with $n \geq 2$. Let $S1, S2, \dots, Sn$ and T be substitutions, then:

1. CASE E OF EITHER $L1$ THEN $S1$ OR $L2$ THEN $S2$... OR Ln THEN Sn END \cong
 SELECT $E \in \{L1\}$ THEN $S1$ OR $E \in \{L2\}$ THEN $S2$ OR ... OR $E \in \{Ln\}$ THEN Sn ELSE skip
 END
2. CASE E OF EITHER $L1$ THEN $S1$ OR $L2$ THEN $S2$... OR Ln THEN Sn ELSE T END =
 SELECT $E \in \{L1\}$ THEN $S1$ OR $E \in \{L2\}$ THEN $S2$ OR ... OR $E \in \{Ln\}$ THEN Sn ELSE T END

Typing rule

In a CASE substitution, the expression as well as the lists of branch constants EITHER and OR must all be of the same simple type: integer type, boolean type, deferred type or enumerated type.

Description

The CASE substitution is used to define for a program, various possible behaviors depending on the value of an expression. Each EITHER and OR branch is made up of a non empty list of constants. The values of constants in the set of branches must be distinct two by two. If the value of the expression belongs to one of the branches, then the substitution of this branch is executed. If not, the substitution in the ELSE branch is applied, if this latter branch is absent, it will by default perform the identity substitution. The behavior of this substitution is therefore deterministic and always feasible.

Example

```

CASE x / 10 OF
    EITHER 0 THEN
        x := 0
    OR 2, 4, 8 THEN
        x := 1
    OR 3, 9 THEN
        x := 2
    ELSE
        x := -1
    END
END

```

6.10 Unbounded choice Substitution

Syntax

Any_substitution ::= "ANY" Ident⁺; "WHERE" *Predicate* "THEN" *Substitution* "END"

Definition

Let X be a non empty list of variables that are distinct two by two, S substitution and P and R two predicates, then :

$$[\text{ANY } X \text{ WHERE } P \text{ THEN } S \text{ END}] R \Leftrightarrow \forall X. (P \Rightarrow [S]R)$$

Scope

In an ANY X WHERE P THEN S END substitution, the scope of the list of identifiers X is the predicate P and the substitution S , and they are accessible in read-only mode in substitution S .

Restriction

- The substitution ANY is not an implementation substitution
- The identifiers introduced by ANY must be distinct two by two.
- The variables X introduced by the ANY X WHERE P THEN S END substitution must be typed by an abstract data typing predicate (refer to 3.1 *Typing foundations*), located in a list of conjunctions at the highest level of syntax analysis in P . These variables cannot be used in P before they have been typed.

Description

The ANY L WHERE P THEN S END substitution allows the use in the substitution S of the abstract data declared in list L and which verify the predicate P .

The data in list L must be distinct two by two. The predicate P must start by typing the L data used in P or in S using the abstract data typing predicates. If a number of values satisfy predicate P , the substitution does not specify which one is effectively chosen. It then defines a non deterministic behavior. The abstract data from list L will then be accessible in read mode, but not in write mode in S , as these are non local variables but abstract data defined by the predicate P .

Example

```

ANY r1, r2 WHERE
  r1 ∈ NAT ∧
  r2 ∈ NAT ∧
  r12 + r22 = 25
THEN
  SumR := r1 + r2
END

```

6.11 Local Definition Substitution

Syntax

```
Let_substitution ::=
  "LET" Ident+; "BE"
  ( Ident "=" Expression )+ ^
  "IN" Substitution "END"
```

Definition

Let x_1, \dots, x_n be a non empty list of identifiers that are distinct two by two, E_1, \dots, E_n a list of expressions, S a substitution, then:

```
LET  $x_1, \dots, x_n$  BE  $x_1 = E_1 \wedge \dots \wedge x_n = E_n$  IN  $S$  END =
  ANY  $x_1, \dots, x_n$  WHERE  $x_1 = E_1 \wedge \dots \wedge x_n = E_n$  IN  $S$  END
```

Restrictions

- The substitution LET is not an implementation substitution
- The identifiers introduced in a given substitution LET must be distinct two by two.
- Each identifier x_i introduced in a substitution LET L BE P IN S END must be defined once and only once with an abstract data typing predicate.
- Only identifiers x_i introduced after the reserved keyword LET can appear in the left hand part of predicates introduced by the reserved keyword BE

Scope

In a LET L BE P IN S END substitution, the identifiers in list L are accessible in the left hand part of the typed predicates that make up predicate P , but not in the expressions in the right hand part and they are accessible in read-only mode in substitution S .

Description

The LET L BE P IN S END substitution introduces a list of abstract data L , the value of which is given by predicate P and which can be used in read mode in substitution S .

The list L must be non empty. It comprises identifiers that are distinct two by two and which establish the predicate P . This predicate comprises a list of conjunctions matching $x_i = E_i$ where x_i is an identifier in list L and where E_i is an expression. Each identifier x_i must be defined once and only once in P , its associated expression E_i must not use any of the identifiers from L . To define identifiers that are dependent on other identifiers, simply use two nested LET substitutions. The identifiers introduced by LET may be used in read-only mode in substitution S .

Example

```
LET  $r1, r2$  BE
   $r1 = (Var1 + Var2) / 2 \wedge$ 
   $r2 = (Var1 - Var2) / 2$ 
IN
   $SumR := r1 + r2 \parallel$ 
   $DifferenceR := r1 - r2$ 
END
```

6.12 Becomes Element of Substitution

Operator

$:\in$ Becomes element of

Syntax

Becomes_elt_substitution ::= (Ident⁺)⁺ ":\in" Expression

Definition

Let E be an expression representing a set, X a list of modifiable non empty variables and Y a list of intermediate variables with as many elements as X but not present in X and E . Then :

$$X : \in E = \text{ANY } Y \text{ WHERE } Y \in E \text{ THEN } X := Y \text{ END}$$

Typing rule

In the $x_l : \in E$ substitution, if x_l is a type t_l then E must be a $\mathbb{P}(t_l)$ set type. In the $x_1, \dots, x_n : \in E$ substitution, if (x_1, \dots, x_n) is a Cartesian product type $(t_1 \times \dots \times t_n)$ then E must be a $\mathbb{P}(t_1 \times \dots \times t_n)$ set type.

Restrictions

- The substitution “Becomes element of” is not an implementation substitution
- The identifiers introduced in a substitution “Becomes an element of” must be distinct two by two.

Description

The “becomes an element of” substitution is used to replace variables with values that belong to a set. The variables must be distinct two by two. If the set has several values, the substitution does not specify which one is effectively chosen, it then defines a non deterministic behavior.

Examples

```
i1 :∈ INT ;
b1 :∈ BOOL ;
x1 :∈ -10 .. 10 ;
y1, y2 :∈ {1, 3, 5} × NAT
```

6.13 Becomes such that Substitution

Operator

`:()` Becomes such that

Syntax

Becomes_such_that_substitution ::= (Ident⁺)⁺ ":" "(" Predicate ")"

Definition

Let P be a predicate, X a list of modifiable variables that are distinct two by two, Y a list of intermediate variables with as many elements as X but that are not in X and P , then:

1. $X:(P) = \text{ANY } Y \text{ WHERE } [X := Y]P \text{ THEN } X := Y \text{ END}$

With a variable y from X . The notation $y\$0$ may be used in P . It represents the value of variable y before the application of the “becomes such that” substitution. Then:

2. $X:(P) = \text{ANY } Y \text{ WHERE } [X, y\$0 := Y, y]P \text{ THEN } X := Y \text{ END}$

Restrictions

- The substitution “Becomes such that” is not an implementation substitution.
- Substitution variables X such as $X:(P)$ must be accessible in write mode.
- In the expression $X:(P)$, the variable list X must be typed in the predicate P with the help of abstract data typing predicates located in a conjunction list at the highest level of the syntax analysis of P .

Typing rule

In the expression $X:(P)$, if x is a variable in the list X , then predicate P must type x if it is not yet typed, using an abstract data typing predicate. This case may only occur when x is a local variable declared in a VAR substitution or by an operation formal output parameter.

Description

The “becomes such that” substitution is used to replace variables with values that satisfy a given predicate. The variables must be distinct two by two. If a number of values satisfy the predicate, the substitution does not specify which one will effectively be chosen, its behavior will then be non deterministic.

The value prior to the substitution of a variable y from X may be referenced by $y\$0$ in predicate P . This possibility is a notational simplification that avoids introducing an intermediate variable into an ANY substitution.

Examples

```
x : (x ∈ INTEGER ∧ x > -4 ∧ x < 4) ;
a, b : (a ∈ INT ∧ b ∈ INT ∧ a2 + b2 = 25) ;
y : (y ∈ NAT ∧ y$0 > y)
```

This last substitution can be written without using the \$0 notation, as follows:

```
ANY y2 WHERE
  y2 ∈ ℤ ∧ y > y2
THEN
  y := y2
END
```

6.14 Local Variable Substitution

Operator

VAR Local variable

Syntax

Var_substitution ::= "VAR" Ident⁺ "IN" *Substitution* "END"

Definition

Let X be a list of variables that are distinct two by two, S a substitution and P a predicate, then:

$$[\text{VAR } X \text{ IN } S \text{ END}]P \Leftrightarrow \forall X.S[P]$$

Scope

In a VAR L IN S END substitution, the identifiers in the L list are accessible in read and in write mode for substitution S .

Restrictions

- The substitution VAR is not an abstract machine substitution
- Variables introduced by a substitution VAR must be distinct two by two.
- In an implementation, local variables must be initialized before being read.

Description

The VAR L IN S END substitution introduces a non empty list of local variables L that are distinct two by two. These local variables may be used in substitution S . They are typed when they are first used in S (in the order of scanning S). The local variables must be initialized before they are read. The keywords IN and END bracket the substitution S like a BEGIN END block substitution.

Examples

```
VAR varLoc1, varLoc2 IN
  varLoc1 := x1 + 1 ;
  varLoc2 := 2 × varLoc1 ;
  x1 := varLoc2
END
```

6.15 Sequencing Substitution

Operator

$;$ Sequencing

Syntax

$Sequence_substitution ::= Substitution ";" Substitution$

Definition

Let S and T be substitutions and P a predicate, then:

$$[S ; T]P \Leftrightarrow [S][T]P$$

Which means that the predicate obtained by applying the sequence substitution $S ; T$ on predicate P is the predicate obtained by applying S to the result of the application of T on P .

Restriction

1. The substitution “Sequencing” is not an abstract machine substitution.

Description

The sequencing substitution corresponds to the execution in sequence of two substitutions.

Example

$z := x ; x := y ; y := z$

In the example above, the values of data items x and y are swapped.

6.16 Operation Call Substitution

Operator

← Operation call

Syntax

$Substitution_call ::= [(Ident^{+;})^{+;}, "\leftarrow"] Ident^{+;}, [(" Expression^{+;}, ")"]$

Definitions

1. Let op be an operation (non local or local) without an output parameter and without an input parameter, defined by $op = S$, then the meaning of an op call is:

$$[op]P \Leftrightarrow [S]P$$

2. Let op be an operation (non local or local) without an output parameter and with input parameters, defined by $op(X) = S$ where X is a list of identifiers that designate the formal input parameters for op , and with E a list of expressions that represent the effective input parameters for op , then the meaning of an $op(E)$ call is:

$$[op(E)]P \Leftrightarrow [X := E][S]P$$

3. Let op be an operation (non local or local) with output parameters and no input parameter, defined by $Y \leftarrow op = S$, where Y is a list of identifiers that refers to the formal output parameters for op , and with R a list of identifiers, possibly renamed, designating the effective output parameters from op , then the meaning of an $R \leftarrow op$ call is:

$$[R \leftarrow op]P \Leftrightarrow [S][R := Y]P$$

4. If op is an operation (non local or local) with output parameters and with input parameters, defined by $Y \leftarrow op(X) = S$, the meaning of an $R \leftarrow op(E)$ call is:

$$[R \leftarrow op(E)]P \Leftrightarrow [X := E][S][R := Y]P$$

Typing rule

In the $op(E)$ and $R \leftarrow op(E)$ operation calls, E is a list of expressions whose type must be identical to the type of the input parameters used for the op operation, defined in the specification which declares this operation.

In the $R \leftarrow op$ and $R \leftarrow op(E)$ operation calls, R is a list of data names whose type must be identical to the type of the output parameters from the op operation defined in the specification which declares this operation.

Restriction

1. A variable must not be used several times as return parameter for a given operation. For example, the operation call $x, x \leftarrow op$ is forbidden

Description

The operation call substitution function is used to apply the substitution of an operation (non local or local), by replacing the formal parameters with effective parameters. Effective input parameters are expressions and effective output parameters are names referring to data that can be modified.

The operation call is defined in four different forms, depending on the presence of input and output parameters.

Examples

```
op1 ;  
op2 (x0 + 1, TRUE) ;  
res1, res2 ← op3 ;  
res, flag ← op4 (x0)
```

6.17 While Loop Substitution

Syntax

```
While_substitution ::=
  "WHILE" Condition "DO" Instruction
  "INVARIANT" Predicate
  "VARIANT" Expression
  "END"
```

Definition

Let P be a predicate, S a substitution, I and R predicates and V an expression. If X represents the list of free variables that appears in S and I and n is a fresh variable, i.e. one that is not free in V , I , P and S , then :

$$\begin{aligned}
 [\text{WHILE } P \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } V \text{ END}]R &\Leftrightarrow \\
 I &\wedge \\
 \forall X. (I \wedge P \Rightarrow [S]I) &\wedge \\
 \forall X. (I \Rightarrow V \in \mathbb{N}) &\wedge \\
 \forall X. (I \wedge P \Rightarrow [n := V] [S](V < n)) &\wedge \\
 \forall X. (I \wedge \neg P \wedge R) &
 \end{aligned}$$

Typing rule

In a `WHILE P DO S INVARIANT I VARIANT V END` substitution, the variant V must be an integer type.

Restriction

1. The substitution “Sequencing” is neither an abstract machine substitution nor a refinement substitution

Description

The `WHILE` substitution is used to perform a “while” loop . The `WHILE P DO S INVARIANT I VARIANT V END` substitution performs the S instruction as long as condition P remains true. A “while” loop must end after a finite number of iterations.

I is the invariant in the loop. It is a predicate that gives the properties of the variables used in the loop. The loop invariant allows proving that at each step, the loop is possible and that on the output, the result produced is available. In the loop invariant, it is possible to designate the value of a variable y for the abstraction of the component using notation $y\$0$.

V is the loop variant. It is an integer expression used to prove that the “while” loop ends. To do this, it is necessary to prove that V is a positive integer expression, that strictly decreases at each iteration.

In the most general case, the `WHILE` substitution may be preceded, in sequence, by an instruction that initialized the variables used in the loop. It will then take the form: $T ; \text{WHILE } P \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } V \text{ END}$ where T is the initialization instruction.

Examples

```
BEGIN
  varLoc := var1 ;
  cpt := 0
END ;
WHILE cpt < 5 DO
  varLoc := varLoc + 1 ;
  cpt := cpt + 1
INVARIANT
  cpt ∈ NAT ∧
  cpt ≤ 5 ∧
  varLoc ∈ NAT ∧
  varLoc = var1 + cpt
VARIANT
  5 - cpt
END
```

6.18 Simultaneous Substitution

Operator

\parallel Simultaneous substitution

Syntax

Simultaneous_substitution ::= Substitution "||" Substitution

Definition

The simultaneous substitution is defined inductively from certain properties and in relation to other language substitutions. Let $S1, S2, S3$ and T be substitutions, x and y of variables, X a list of identifiers, E an expression, $I1$ and $I2$ lists of constants, $P1, P2$ and R predicates, then:

Properties of the simultaneous substitution:

1. $S1 \parallel S2 = S2 \parallel S1$
2. $S1 \parallel (S2 \parallel S3) = (S1 \parallel S2) \parallel S3$

Defining the simultaneous “becomes equal” substitution:

3. $x := E \parallel y := F = x, y := E, F$
4. $[x := E \parallel x := F] R \Leftrightarrow \forall x'. ((x' = E \wedge x' = F) \Rightarrow [x := x'] R)$

Simultaneous substitution behavior in relation to other substitutions:

5. $\text{skip} \parallel S = S$
6. $X:(P) \parallel S = \text{ANY } Y \text{ WHERE } [X := Y]P \text{ THEN } X := Y \parallel S \text{ END}$
7. $X:\in E \parallel S = \text{ANY } Y \text{ WHERE } Y \in E \text{ THEN } X := Y \parallel S \text{ END}$
8. $\text{CHOICE } S \text{ OR } T \text{ END} \parallel U = \text{CHOICE } S \parallel U \text{ OR } T \parallel U \text{ END}$
9. $\text{PRE } P \text{ THEN } S \text{ END} \parallel T = \text{PRE } P \text{ THEN } S \parallel T \text{ END}$
10. $\text{ASSERT } P \text{ THEN } S \text{ END} \parallel T = \text{ASSERT } P \text{ THEN } S \parallel T \text{ END}$
11. $\text{BEGIN } S \text{ END} \parallel T = \text{BEGIN } S \parallel T \text{ END}$
12. If no elementary variable from X is free in T , then:
 $\text{ANY } X \text{ WHERE } P \text{ THEN } S \text{ END} \parallel T = \text{ANY } X \text{ WHERE } P \text{ THEN } S \parallel T \text{ END}$
13. If no elementary variable from X is free in T , then:
 $\text{SELECT } X \text{ THEN } S1 \text{ WHEN } P2 \text{ THEN } S2 \text{ END} \parallel T = \text{SELECT } P1 \text{ THEN } S1 \parallel T \text{ WHEN } P2 \text{ THEN } S2 \parallel T \text{ END}$
14. If no elementary variable from X is free in T , then:
 $\text{LET } X \text{ BE } P1 \text{ IN } S1 \text{ END} \parallel T = \text{LET } X \text{ BE } P1 \text{ IN } S1 \parallel T \text{ END}$
15. $\text{IF } P1 \text{ THEN } S1 \text{ ELSE } S2 \text{ END} \parallel T = \text{IF } P1 \text{ THEN } S1 \parallel T \text{ ELSE } S2 \parallel T \text{ END}$
16. $\text{CASE } E \text{ OF EITHER } I1 \text{ THEN } S1 \text{ OR } I2 \text{ THEN } S2 \text{ END} =$
 $\text{CASE } E \text{ OF EITHER } I1 \text{ THEN } S1 \parallel T \text{ OR } I2 \text{ THEN } S2 \parallel T \text{ END}$
17. If no elementary variable from X is free in T , then:
 $\text{VAR } X \text{ THEN } S \text{ END} \parallel T = \text{VAR } X \text{ THEN } S \parallel T \text{ END}$

Restrictions

1. The simultaneous substitution is not an implementation substitution
2. Let S_x and T_y be two substitutions which modify the variable series X and Y . Then, it

is necessary that the variable list X and Y are differ two by two.

3. It is forbidden to call simultaneously two write operations of the same *included* machine instance. Indeed, even if each operation call preserves the invariant of the *included* machine instance, it would be possible that two simultaneous operation calls break this invariant.
4. In a local operation specification (see section 7.24, *LOCAL_OPERATIONS Clause*), it is illegal to call two parallel operations for the same *included* machine instance.

Description

Simultaneous substitution corresponds to the simultaneous execution of two substitutions. The simultaneous character shows that the substitutions must be able to be performed independently of each other. Simultaneous substitution is commutative and associative. It is not defined globally, but in relation to each of the other substitutions in the language. The following special cases should be noted:

- A number of “becomes equal” substitution grouped in a simultaneous substitution corresponding to one multiple “becomes equal” substitution that affects the set of variables. The rules for multiple “becomes equal” substitution therefore also apply when writing simultaneous substitutions. For this reason, it is illegal to use parallel substitutions that modify the same variables.
- It is illegal to call two parallel operations of the same *included* machine instance, for if these operations modify the same variables of the *included* machine instance, then it would be possible to break its invariant.

Example

```
x := y ||
y := x
```

In the example above, the values of data items x and y are swaped.

7. COMPONENTS

7.1 Abstract Machine

Syntax

```

Machine_abstract ::=
    "MACHINE" Header
    Clause_machine_abstract*
    "END"

Clause_machine_abstract ::=
    Clause_constraints
    | Clause_sees
    | Clause_includes
    | Clause_promotes
    | Clause_extends
    | Clause_uses
    | Clause_sets
    | Clause_concrete_constants
    | Clause_abstract_constants
    | Clause_properties
    | Clause_concrete_variables
    | Clause_abstract_variables
    | Clause_invariant
    | Clause_assertions
    | Clause_initialization
    | Clause_operations

```

Description

An abstract machine is a component that defines in different clauses, data and its properties as well as operations. An abstract machine is the specification of a B module. It comprises a header and some clauses. The order of the clauses in a component is not fixed. The description of clauses is given in the table below.

Clause	Description
CONSTRAINTS	Definition of the type and properties of formal scalar parameters
SEES	List of instances of <i>seen</i> machines
INCLUDES	List of instances of <i>included</i> machines
PROMOTES	List of promoted operations of instances of <i>included</i> machines
EXTENDS	List of instances of <i>extended</i> machines
USES	List of instances of <i>used</i> machines
SETS	List of deferred sets and definition of enumerated sets
CONCRETE_CONSTANTS	List of concrete constants
ABSTRACT_CONSTANTS	List of abstract constants
PROPERTIES	Type and properties of machine constants
CONCRETE_VARIABLES	List of concrete variables
ABSTRACT_VARIABLES	List of abstract variables
INVARIANT	Type and properties of variables
ASSERTIONS	Definition of properties that are deduced from the invariant
INITIALISATION	Initialization of variables
OPERATIONS	List and definition of specific operations

Restrictions

1. A clause may only appear at most once in an abstract machine.
2. If one of the `CONCRETE_CONSTANTS` or `ABSTRACT_CONSTANTS` clauses is present, then the `PROPERTIES` clause must be present.
3. If one of the `CONCRETE_VARIABLES` or `ABSTRACT_VARIABLES` clauses is present, then the `INVARIANT` and `INITIALISATION` clauses must be present.

Use

An abstract machine may be referenced in the `REFINES` clause of a refinement, to declare that the refinement refines this abstract machine or in the component visibility clauses, (`IMPORTS`, `SEES`, `INCLUDES`, `EXTENDS`, or `USES` clauses), to indicate that the component *imports*, *sees*, *includes*, *extends*, or *uses* an instance of this abstract machine.

Visibility tables

Let M_A be an abstract machine. The visibility table below specifies the access mode for each component of M_A (data or operation), in the clauses of M_A .

For example, we may observe that a concrete variable may be read in the `INVARIANT` and `ASSERTIONS` clauses and may be read and written, i.e. modified in the clause `INITIALISATION` clause or in an operation.

The term “specific operation” on a component refers to an operation whose body is defined in the component, within the `OPERATIONS` clause.

Clauses of M_A	<code>CONSTRAINTS</code>	Parameters in <code>INCLUDES / EXTENDS</code>	<code>PROPERTIES</code>	<code>INVARIANT / ASSERTIONS</code>	<code>INITIALISATION / OPERATIONS</code>
Formal parameters	read	read		Read	read
Sets, enumerated set elements, concrete constants		read	read	Read	read
Non homonymous abstract constants		read	read	Read	read
Non homonymous concrete variables				Read	read/write
Non homonymous abstract variables				Read	read/write
Specific operations (not promoted)					

7.2 Header

Syntax

Header ::= Ident ["(" Ident⁺ ")"]

Description

A component header defines the name of the component and the list of its parameters.

A component may be followed by a list of identifiers representing the formal component parameters. The formal parameters of a component remain unchanged in the component refinements. They are used to set the parameters of the different instances of the abstract machine. A formal parameter may either be a deferred set that is a new scalar type (refer to section 7.13 *The SETS Clause*), or a scalar (refer to 7.5 *The CONSTRAINTS Clause*).

Restrictions

1. The name of a component must be unique in the project.
2. The declaration of formal parameters must remain identical in all the components of a module.
3. The name of set parameters must not comprise any lowercase character.
4. The name of scalar parameters must comprise at least one lowercase character.

Use

Let M_A be a parametered abstract machine. On the *inclusion* or *importing* of an instance of M_A , the formal parameters of M_A are instantiated by effective parameters. The properties of the formal parameters of M_A are declared in the abstract machine CONSTRAINTS clause.

The formal parameters may be used in read mode, in the INVARIANT and ASSERTIONS, INCLUDES clauses (as an effective parameter of the *included* abstract machine), as well as in the INITIALISATION and OPERATIONS clauses.

It should be noted that the formal parameters cannot be used in the PROPERTIES clause.

7.3 Refinement

Syntax

```

Refinement ::=
    "REFINEMENT" Header
    Clause_refines
    Clause_refinement*
    "END"

Clause_refinement ::=
    Clause_sees
    | Clause_includes
    | Clause_promotes
    | Clause_extends
    | Clause_sets
    | Clause_concrete_constants
    | Clause_abstract_constants
    | Clause_properties
    | Clause_concrete_variables
    | Clause_abstract_variables
    | Clause_invariant
    | Clause_assertions
    | Clause_initialization
    | Clause_operations

```

Description

A refinement is a component that refines an abstract machine or another refinement (refer to 8.2 *B Module*).

The USES and CONSTRAINTS clauses are illegal in a refinement.

Restrictions

1. A clause may only appear at most once in a refinement.

Use

A refinement may be used in a REFINES clause in another refinement, to declare that this other refinement refines the first one.

Visibility tables

Let M_n be a refinement. The visibility table below specifies the access mode for each constituent of M_n , in the clauses of M_n .

Clauses of M_n Constituents of M_n	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS
Formal parameters	Read		Read	Read
Sets, enumerated set elements, concrete constraints	Read	Read	Read	Read
Non homonymous abstract constants	Read	Read	Read	Read
Non homonymous concrete variables			Read	Read/write
Non homonymous abstract variables			Read	Read/write
Specific operations (not promoted)				

Let M_{n-1} be the component refined by M_n . The visibility table below specifies the access mode for each constituent of M_{n-1} disappearing in M_n , in the clauses of M_n . Here, only the abstract data from M_{n-1} and they disappearing in M_n are of interest as the other data is preserved in M_n .

In the case of initialization and of operations, there is a difference between the visibility in the assertion substitution predicates (refer to section 6.5 *Assertion Substitution*) compared with the rest of the substitutions.

Clauses of M_n Constituents of M_{n-1}	Parameters of INCLUDES /EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS	
				Substitutions	Predicates of ASSERT
Abstract constants that disappear in M_n		read	read		read
Abstract variables that disappear in M_n			read		read

7.4 Implementation

Syntax

```

Implementation ::=
    "IMPLEMENTATION" Header
    Clause_refines
    Clause_implementation*
    "END"

Clause_implementation ::=
    Clause_sees
    | Clause_imports
    | Clause_promotes
    | Clause_extends_B0
    | Clause_sets
    | Clause_concrete_constants
    | Clause_properties
    | Clause_values
    | Clause_concrete_variables
    | Clause_invariant
    | Clause_assertions
    | Clause_initialization_B0
    | Clause_operations_B0

```

Description

An implementation is a component that constitutes the last refinement of an abstract machine (refer to 8.2 B Module).

Two new clauses may appear in an implementation: the IMPORTS clause and the VALUES clause. The IMPORTS clause creates modules concrete instances in a project. The VALUES clause is used to assign a value to the deferred sets and to the concrete constants.

The EXTENDS clause corresponds to the IMPORTS clause in an implementation. In an abstract machine or in a refinement, the EXTENDS clause corresponds to the INCLUDES clause.

The INITIALISATION and OPERATIONS clause differ from an implementation and an abstract machine or a refinement. In an implementation, these clauses are made up of concrete expressions or substitutions (refer to section 7.24 LOCAL_OPERATIONS Clause).

The CONSTRAINTS, INCLUDES, USES, ABSTRACT_CONSTANTS and ABSTRACT_VARIABLES (or VARIABLES) clauses are illegal in an implementation.

Restriction

1. A clause cannot appear more than once in an implementation.

Use

Visibility tables

Let Mn be an implementation. The visibility table below specifies the access mode for each component of Mn , in the clauses of Mn . In the initialization and the operations, a difference is made between the use of components in the non translated parts: the variant and the invariant of the WHILE loop and the assertion of an ASSERT substitution and in the translated parts: the rest of B0 instructions.

Clauses in M_n	Parameters of IMPORTS /EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS	
					Instructions	loops variants and invariants, predicate ASSERT
Constituents of M_n						
Formal parameters	Read			Read	Read	Read
Enumerated sets, elements in enumerated sets	Read	Read	Read	Read	Read	Read
Deferred sets, concrete constants	Read	Read	Write	Read	Read	Read
Concrete variables				Read	Read/Write	Read
Specific operations						
Definitions	Read	Read	Read	Read	Read	Read

Let M_{n-1} be the abstraction of M_n . The visibility table below shows the access mode for each component of M_{n-1} that disappear in M_n , in the clauses of M_n .

Clauses of M_n	Parameters of IMPORTS /EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS	
				Instructions	loops variants and invariants, predicate ASSERT
Constituents of M_{n-1}					
Abstract constants		Read	Read		Read
Abstract variables			Read		Read

7.5 The CONSTRAINTS Clause

Syntax

Clause_constraints ::= "CONSTRAINTS" *Predicate*^{+"^"}

Description

The CONSTRAINTS clause is used to type the scalar parameters of the abstract machine and to express complementary properties, also called constraints, that apply to these parameters.

The abstract machine parameters are of two kinds:

- scalar parameters: the name of a scalar parameter is an identifier that must contain at least one lowercase character. Its type may be \mathbb{Z} , BOOL or a machine set parameter (see below),
- set parameters: the name of a set parameter is an identifier that must not contain a lowercase character. It represents a deferred set, equipotent to an interval of integers. Set parameters form new types that may be used to type machine scalar parameters.

Restrictions

1. Each abstract machine scalar parameter must be typed by a machine parameter typing predicate (refer to section 3.1 *Typing foundations*) located at the first level of a list of conjunctions.
2. Each scalar parameter must be typed prior to use (refer to section 3.1 *Typing foundations*).
3. The set parameters do not require typing.

Use

Formal set parameters define new types like deferred sets or enumerated sets (refer to section 7.13 *The SETS*), that take their name. The types of the different set parameters are incompatible. It is therefore illegal, for example, to express that a set parameter is contained in another one.

Example

In the example below, $p1$ is an integer parameter, $p2$ is a Boolean parameter, and $p3$ is a parameter that belongs to the *ENS1* set parameter. In addition, *ENS1* and *ENS2* are set parameters.

```
MACHINE
  MA ( p1, p2, p3, ENS1, ENS2 )
CONSTRAINTS
  p1 ∈ INT ∧
  p2 ∈ BOOL ∧
  p1 ∈ ENS1
  ...
END
```

7.6 The REFINES Clause

Syntax

Clause_refines ::= "REFINES" Ident

Description

The REFINES clause contains the name of the refined component (also called abstraction) for the refinement.

7.7 The IMPORTS Clause

Syntax

```

Clause_IMPORTS ::=
    IMPORTS ( [Ident"."]Ident [ "(" Instanciacion_B0+ ";" "]" ] )+ ;
Instanciacion_B0 :=
    Term
    | Number_set_B0           (refer to section 3.5)
    | BOOL
    | Interval                 (refer to section 5.7)

```

Description

The *import* link between an implementation and an abstract machine instance is a composition link. The implementation creates the *imported* abstract machine instance to use its data and operations to implement its own data and operations. The module of the implementation is therefore the father of the *imported* module instance into the project *import* graph (refer to section 8.3 *IMPORTS link*). This implementation is the only implementation of the project that has the right to modify the *imported* machine instance variables .

Restrictions

1. The IMPORTS clause identifiers must refer to abstract machines.
2. The renamed identifiers have no more than one renaming prefix.
3. Each machine name must be followed by an effective parameter list with the same number of parameters than the *imported* machine.
4. If a concrete constant of a machine instance *imported* by an implementation is homonymous to a concrete constant of the implementation, then both homonymous constants refer to the same data and must be of the same type.
5. If a concrete variable of a machine instance *imported* by an implementation is homonymous to a concrete variable of the implementation, then both homonymous variables refer to the same data and must be of the same type.
6. A machine instance must not be *imported* more than once in a project (refer to 8.3 Rule n°1 on IMPORT links).
7. A complete project must contain one, and only one, developed module that is never *imported* in the project (refer to 8.3 Rule n°2 on IMPORT links).
8. A component must not have more than one link to the same machine instance. For example, an implementation cannot *see* (refer to 7.8 *Clause SEES*) and *import* the same machine instance (refer to 8.3 Rule n°7 on dependency rules).
9. There must not be any cycle in the dependency graph of a project (refer to 8.3 Rule n°8 on dependency rules).

Use

The IMPORTS clause contains the declaration of the list of *imported* machine instances. These contain either the name of the machine only, that references the machine instance with no renaming, or the name of the machine preceded by a renaming prefix that refers to the instance of the renamed machine. If an *imported* machine instance has parameters,

the effective parameters of the machine must be provided to instantiate the instance of the formal parameters of the new machine instance. The parameters of an abstract machine are described in the `CONSTRAINTS` clause (refer to 7.5 The `CONSTRAINTS`). They may be scalars or sets of scalars. It is necessary to prove that the effective parameters of an *imported* machine instance meet the machine constraints.

Scalar Parameters Instantiation

An effective scalar parameter of an imported machine instance may be:

- a literal Boolean value `TRUE` or `FALSE`,
- an element of an enumerated set of the implementation or a *seen* machine instance (refer to section 7.8 The *SEES Clause*)
- a formal scalar parameter from the implementation,
- a concrete constant of type `Z` or `BOOL` from the implementation or a *seen* machine instance,
- an arithmetical expression formed from formal implementation parameters, concrete constants or *seen* machine instances and literal integers. The arithmetical operators allowed are `'+'`, `'-'`, `'*'`, `'/'`, `mod`, a^b , `succ` and `pred`. It is necessary to prove that each arithmetical sub-expression is in fact defined and that its result belongs to the set of implementable integers `INT` (refer to 3.1 Typing foundations).

Set Parameters Instantiation

The instantiation of a set parameter for an *imported* machine instance may be:

- a formal set parameter of the implementation,
- an deferred or enumerated set of the implementation or a *seen* machine instance,
- a non empty interval with bounds made up of arithmetical expressions similar to those allowed to instantiate scalar parameters.

Example

```
MACHINE
  MA (E0)
CONCRETE_CONSTANTS
  c1
PROPERTIES
  c1 ∈ 0 .. 10
...
END
```

```
MACHINE
  Msees
SETS
  COUL = { Rouge, Vert, Bleu }
CONSTANTS
  c2
PROPERTIES
  c2 ∈ INT
END
```

```
IMPLEMENTATION
  MA_i (E0)
REFINES
  MA
SEES
  Msees
IMPORTS
  Mimports (E0, COUL, c1 .. (c2 + 2))
VALUES
  c1 = 2
...
END
```

```
MACHINE
  Mimp (ENS1, ENS2, ENS3)
...
END
```

Visibility

The formal parameter of *imported* machines are not accessible in the component which *imports*. Sets and concrete constants are accessible in the clauses PROPERTIES, VALUES, INVARIANT, ASSERTIONS and in the body of the initialization and the operations of the machine. Abstract constants are accessible in the clauses PROPERTIES, INVARIANT, ASSERTIONS and in the loop variants and loop invariants and in the predicate of the ASSERT instructions (in operations and initialization). Variables are accessible in read mode in the invariants and assertions. Concrete variables are accessible in read mode in the initialization and operation body. Abstract variables are accessible only in loop variants and loop invariants and in the predicate of the ASSERT instructions (in operations and initialization). It is possible to use *imported* machine operations in the initialization and in the operations of the implementation.

Promotion of operations

The *imported* machine operations can become automatically operations of the *importing* component (refer to section 7.10, *The PROMOTES Clause* and section 7.11, *The EXTENDS Clause*).

Visibility tables

Let M_{A_i} be an implementation that *imports* an M_B machine instance. The visibility table below specified the access mode for each constituent of M_B , in the clauses of M_{A_i} .

Clauses of M_{A_i} Constituent of M_B	Parameters of IMPORTS /EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS		LOCAL OPERATIONS
					Instructions	loops variants and invariants, ASSERT predicate	
Formal parameters							
Sets, enumerated set elements, concrete constants		read	read	read	read	read	read
Abstract constants		read		read		read	read
Concrete variables				read	read/non write	read	read/write
Abstract variables				read		read	read/write
Operations					read/write		read/write

7.8 The SEES Clause

Syntax

$Clause_sees ::= "SEES" (Ident^{*})^{*},"$

Description

The SEES link is used to reference within a component, an abstract machine instance *imported* into another branch of the project, to access its constituents (sets, constants and variables) without modifying them.

Restrictions

1. A *seen* machine instance (SEES clause) in a project must be *imported* in the project (refer to 8.3 Rule n°3 of SEES links).
2. If a machine instance is *seen* by a component of a developed module, then the refinements of this component must also *see* this instance (refer to 8.3 Rule n°4 on SEES links).
3. A module component cannot *see* a machine instance that belongs to the module *import* sub-graph (refer to 8.3 Rule n°5 on SEES links).
4. If a component *sees* a machine instance MA then it cannot *see* a machine instance that belongs to the *imports* sub-graph of MA (refer to 8.3 Rule n°6 on SEES links).
5. A component must not have more than one link to the same machine instance. For example, an implementation cannot *see* and *import* the same machine instance (refer to 8.3 Rule n°7 on dependency links).
6. Cycles are forbidden in the dependency graph of a project (refer to 8.3 Rule n°8 on dependency links).

Use

The list of *seen* machine instances is made up of machine names, possibly renamed. The meaning of this renaming is given in the next paragraph. If a machine has parameters the effective parameters of the machine must not be provided, as these will only be provided in the INCLUDES and IMPORTS clauses that create the machine instances, but not in the SEES clause that only references an already existing machine instance.

SEES and renaming

When machine M_A *sees* a machine instance M_B , the name of the instance that is actually *seen* is built from the name of the machine *seen*, preceded by any successive renaming of the instance of M_B , that occurs in its *imports* tree, starting from the first common ancestor of M_A and M_B .

Considering the following example:

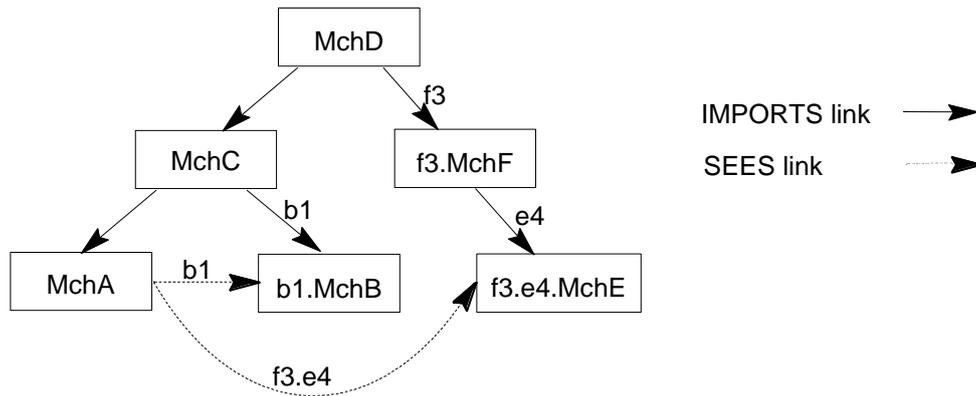


Figure 1: Simple example of SEES and renaming

The figure above shows the dependency graph between machine instances of a project. *MchA* sees the instances *b1.MchB* and *f3.e4.MchD*. In practice, *MchC* is the first common ancestor of *MchA* and *MchB*. From *MchC*, *MchB* is *imported* with the renaming prefix *b1*. In the same way, from *MchD*, *MchE* is *imported* with the successive renaming prefixes *f3* then *e4*.

Example 2 is build as an extension of example 1. Now *MchD* also *imports* a new renamed instance of *MchC*, *c2.MchC*. The graph below shows the instances that are effectively *seen* by the new instance of *MchA* created transitively by this renaming procedure.

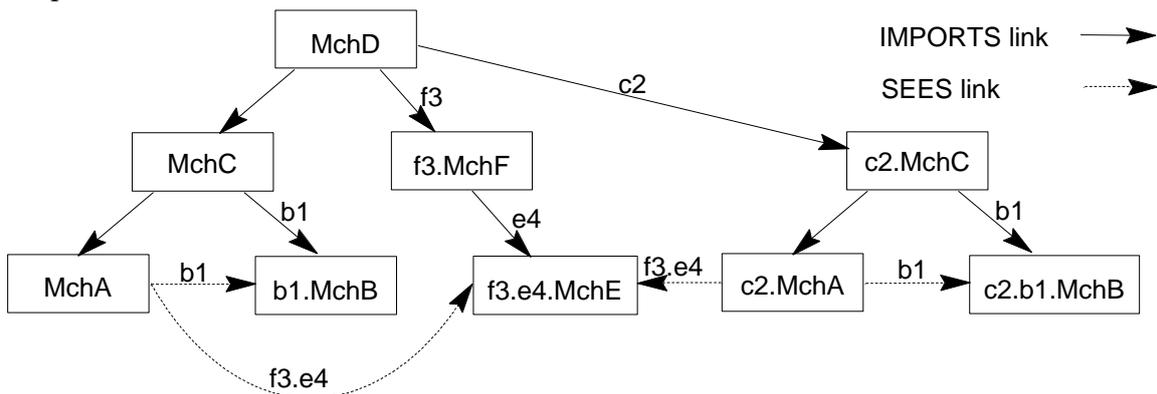


Figure 2: More complex example of SEES and renaming

The new instance of *MchA* noted *c2.MchA* sees the instances *c2.b1.MchB* and *f3.e4.MchE*. In practice, *c2.MchC* is the first common ancestor of *c2.MchA* and *c2.b1.MchB*. From *c2.MchC*, *c2.b1.MchB* is *imported* with the renaming prefix *b1*. In the same way, *MchD* is the common ancestor of *c2.MchA* and *MchE*. From *MchD*, *MchE* is *imported* with the renaming prefix *f3.e4*.

Visibility

Let M_A be an abstract machine or a refinement that *sees* a machine instance M_B . The formal parameters of M_B are not accessible in M_A . The sets and constants of M_B are accessible in the INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS clauses and in the body of the initialization and operations of the component. The variables are accessible in read mode in the body of the initialization and of the operations. It is possible to use the access operations (that do not modify the variables) for M_B in the initialization and in the operations on M_A .

If the instance of the *seen* machine is renamed, then its variables and its operations are accessible in the machine by prefixing their name with the renaming prefix of the *seen* machine.

Let M_{A_i} be an abstract machine or a refinement that *sees* an instance of machine M_B . The formal parameters of M_B are not accessible in M_{A_i} . The sets and the concrete constants of M_B are accessible in the IMPORTS, EXTENDS, PROPERTIES, VALUES, INVARIANT, ASSERTIONS clauses and in the body of the initialization and the operations of the component. The abstract constants of M_B are also accessible in the PROPERTIES, INVARIANT, ASSERTIONS clauses and in loops variants and invariants, and in the predicates of ASSERT substitutions in operations and the initialization. The concrete variables of M_B are accessible in read mode and in the body of the initialization and operations. The abstract variables of M_B are also accessible in loops variants and invariants, and in the predicates of ASSERT substitutions in operations and initialization. It is possible to use consultation operations (that do not modify the variables) of M_B in the initialization and in the operations of M_{A_i} .

Transitivity

The SEES clause is not transitive. If a component M_1 *sees* a machine M_2 that itself *sees* as machine M_3 , then the components of M_3 are not accessible by M_1 . If they should be, then M_1 should also explicitly *see* M_3 .

Visibility tables

Let M_A be a machine or a refinement that *sees* a machine M_B . The visibility table below specifies for each component of M_B , the access modes that apply in the clauses of M_A .

Clauses of M_A	CONSTRAINTS	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS
Components of M_B					
Formal parameters					
Sets, enumerated set elements, concrete constants		Read	Read	Read	Read
Abstract constants		Read	Read	Read	Read
Concrete variables					Read/non write
Abstract variables					Read/non write
Operations					Read/non write

Let M_{A_i} be an implementation that *sees* an instance of machine M_B . The visibility table below specifies for each component of M_B , the access modes that apply in the clauses of M_{A_i} .

Clauses of M_{A_i} Components of M_B	Parameters of IMPORTS / EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS		LOCAL OPERATIONS
					Instructions	loops variants and invariants, ASSERT predicates	
Formal parameters							
Sets, enumerated set element, concrete constants	Read	Read	Read	Read	Read	Read	Read
Abstract constants		Read		Read		Read	Read
Concrete variables					Read	Read	Read/non write
Abstract variables						Read	Read/non write
Operations					Read/non write		Read/non write

7.9 The INCLUDES Clause

Syntax

```

Clause_includes ::=
    "INCLUDES" ( [Ident.]Ident [ "(" Instantiating+ ")" ] )+
Instantiating :=
    Term
|   Number_set
|   "BOOL"
|   Interval

```

Typing Rule

When an instance of an *included* machine has formal parameters then effective scalar parameters must have the same type as the corresponding formal parameters of the *included* machine and effective set parameters must have a type matching $\mathbb{P}(T)$ where T must be a base type different from STRING.

Restrictions

1. The identifiers of an INCLUDE clause (disregarding the possible renaming prefix) must refer to abstract machines.
2. Renamed identifiers must have only one renaming prefix.
3. Each identifier referring to a machine must be followed by the same number of effective parameters as there is formal parameters.

Description

The INCLUDES clause is used to bring together a component, the components (sets, constants and variables) of machine instances as well as their properties (PROPERTIES and INVARIANT clauses), in order to make up a complex abstract machine using other abstract machines.

Use

The INCLUDES clause comprises the declaration of the list of *included* machines instances. Each instance may be either a machine name, it then refers to the instance with no renaming prefix, or a machine name preceded by a renaming prefix, it then refers to the renamed machine instance. If an included machine instance has parameters, then effective parameters should be supplied, in order to instantiate the formal parameters of the newly created machine instance. Parameters of an included machine are described in the CONSTRAINTS clause (see section 7.5 *The CONSTRAINTS Clause*). Machine parameters are either scalar parameters or set parameters. A Proof Obligation is generated in order to prove that effective parameters meet the constraints of formal parameters.

Instanciation of scalar parameters

An effective scalar parameter of an *included* machine instance has the type \mathbb{Z} , BOOL or *Set*, if *Set* is a deferred set or an enumerated set.

Instanciation of set parameters

An effective set parameter of an *included* machine instance has the type $\mathbb{P}(\mathbb{Z})$, $\mathbb{P}(\text{BOOL})$ or $\mathbb{P}(\text{Set})$, if *Set* is a deferred set or an enumerated set.

Refinement of an *including* component

When the abstraction of a refinement M_{A_r} *includes* a machine instance M_B , then refinement M_{A_r} may *include* M_B again. If M_B is *included* again, then deferred sets, enumerated sets, concrete constants and concrete variables of M_{A_r} that comes from the previous *inclusion* of M_B are glued with those of M_B . Abstract constants and abstract variables of M_{A_r} that comes from the previous *inclusion* of M_B are preserved in M_{A_r} as they are glued with data of M_B .

Implementation of an *including* component

When a MA component *includes* a machine instance MB , then a number of possibilities must be envisaged when writing implementation MA_i in MA . The implementation may *import* the instance of the *included* machine MA . In this case, the components of MB grouped in MA on *inclusion* are implemented in MA_i by applying the same name as those of the instance of the *imported* machine. If not, the implementation may not *import* the instance of the MB machine. It must then implant the components of MB grouped in MA on the *inclusion* either directly, or via the components of instances of machines *seen* or *imported*. The user remains free to choose the breakdown of their choice. In the second case, if no instance of MB is *imported* in the project then MB is called an abstract module. The abstract instance of MB is then created to serve as a specification intermediary and it is abandoned in the development sequence.

Visibility

The formal parameters of *included* machine are not accessible in the component that *includes*. The sets and the constants are accessible in the `PROPERTIES`, `INVARIANT`, `ASSERTIONS` clauses and in the body of the initialization and of machine operations. The variables are accessible in the invariants and the assertions. They are also accessible in read mode in the body of the initialization and of operations. It is possible to use the operations of a machine *included* in the machine initialization and operations.

Transitivity

The `INCLUDES` clause is transitive: if a component M_1 *includes* an instance of machine M_2 that in turn *includes* an instance of machine M_3 , then the sets, the constants, the variables and the properties of M_3 are grouped with those of M_2 that are in turn grouped with those of M_1 . These components are accessible by M_1 . However, the operations of M_3 are not accessible by M_1 . The grouping and access properties are defined for a given number of machines *included* by transitivity.

Grouping data

If a component M_A *includes* instances of machines M_{inc} , then, in relation to the exterior of the component, the set of components (the sets, the constants and the variables) of *included* machines and *included* by transitivity is part of component M_A in the same way as the components that belong to this component. Therefore, if a component M_B *sees* M_A , the sets, the constants and the variables of machines *included* and transitively *included* by M_A are accessible in component M_B according to the same rules as the sets, the constants and the variables of M_B . If a component M_{A_r} refines M_A , the sets, the constants and the variables of machines *included* and transitively *included* by M_A are accessible in component M_{A_r} according to the same rules as the sets, the constants and the specific variables of M_A . Especially, the concrete variables of machines *included* by M_A must all be initialized in M_{A_r} and are accessible in write mode in the body of operations in M_{A_r} .

Promotion of operations

The operations for an *included* machine may automatically become operations on the component that performs the *inclusion*. It is the mechanism used to promote the operation (refer to section 7.10 The PROMOTES and section 7.11 The EXTENDS).

Visibility table

Let M_A be a machine or a refinement that *includes* a machine instance of M_B . The visibility table and the visibility table below specify for each component of M_B , the usage modes that apply in the clauses of M_A .

Clauses of M_A	CONSTRAINTS	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Components of M_B					
Formal parameters					
Sets, enumerated set elements, concrete constants			Read	Read	Read
Abstract constants			Read	Read	Read
Concrete variables				Read	Read/non write
Abstract variables				Read	Read/non write
Operations					Read/write

Note that these visibility links are the same as those that would have occurred if the clauses of the *included* machine were grouped in the corresponding clauses in the machine that causes the *inclusion* (refer to the grouping diagram below).

Equivalent machine

If MA is an abstract machine that *includes* MB , the logically equivalent machine derived from the *inclusion* of these two machines is presented. The name of the equivalent machine, its parameters and its operations are those of MA . Its abstract and enumerated sets, its constants, its properties, its variables, its invariant, its assertions and its initialization respectively are those of MB concatenated with those of MA . In the case of the initialization, note that the equivalent substitution is the sequencing of the initialization of MB then of MA , in practice, as the operations of MB may be called in the initialization of MA , it is necessary for the invariant of MB to have already been established.

```

MACHINE
  MA ( ParamA )
CONSTRAINTS
  ConstrA
INCLUDES
  MB ( Inst )
SETS
  SetsA
CONCRETE_CONSTANTS
  ConcCstA
ABSTRACT_CONSTANTS
  AbsCstA
PROPERTIES
  PropA
CONCRETE_VARIABLES
  ConcVarA
ABSTRACT_VARIABLES
  AbsVarA
INVARIANT
  InvA
ASSERTIONS
  AssertA
INITIALISATION
  InitA
OPERATIONS
  opA1 = SubstA1 ;
  ...
  opAn = SubstAn
END

```

```

MACHINE
  MB ( ParamB )
CONSTRAINTS
  ConstrB

SETS
  SetsB
CONCRETE_CONSTANTS
  ConcCstB
ABSTRACT_CONSTANTS
  AbsCstB
PROPERTIES
  PropB
CONCRETE_VARIABLES
  ConcVarB
ABSTRACT_VARIABLES
  AbsVarB
INVARIANT
  InvB
ASSERTIONS
  AssertB
INITIALISATION
  InitB
OPERATIONS
  opB1 = SubstB1 ;
  ...
  opBm = SubstBm
END

```

⇔

```

MACHINE
  MA ( ParamA )
CONSTRAINTS
  ConstrA

SETS
  SetsB ; SetsA
CONCRETE_CONSTANTS
  ConcCstB , ConcCstA
ABSTRACT_CONSTANTS
  AbsCstB , AbsCstA
PROPERTIES
  PropB  $\wedge$  PropA
CONCRETE_VARIABLES
  ConcVarB , ConcVarA
ABSTRACT_VARIABLES
  AbsVarB , AbsVarA
INVARIANT
  InvB  $\wedge$  InvA
ASSERTIONS
  AssertB ; AssertA
INITIALISATION
  InitB ; InitA
OPERATIONS
  opA1 = SubstA1 ;
  ...
  opAn = SubstAn
END

```

7.10 The PROMOTES Clause

Syntax

`Clause_promotes ::= "PROMOTES" (Ident+".")+"."`

Description

The PROMOTES clause allows a component to promote operations (refer to section 7.23 The OPERATIONS) belonging to machine instances created by the component. These may be instances of *included* machines, if the component is an abstract machine or a refinement, instances of *imported* machines if the component is an implementation. Promoting an operation of a machine instance M_B in a component M_A is equivalent to define in M_A an operation that takes the name of the operation of M_B (possibly preceded by the renaming prefix for M_B , if M_B is renamed), and with the signature and the body of the operation in M_B .

Restrictions

1. The names of the operations promoted by a component must designate operations of *included* machine instances, if the component is an abstract machine or a refinement, or of *imported* machine instances, if the component is an implementation.
2. Each promoted operation of an abstract machine refinement must have the name of an abstract machine operation. Both operations must have the same signature (their formal parameters must have the same name, be in the same order and be of the same types).

Usage

Each name in the PROMOTES list refers to an operation of an *included* machine instance. If the *included* machine instance is renamed, then the name of the operation must be preceded by the renaming prefix of the instance. The name, the signature and the service provided by a promoted operation are identical to the name, the signature and the service provided by the operation that it promotes.

The promoted operations become independent operations of the abstract machine. From the point of view of components that use this machine, nothing distinguishes the promoted operations from operations declared in the OPERATIONS clause.

Unlike operations declared in the OPERATIONS clause, promoted operations may be called in the machine INITIALISATION and OPERATIONS clauses, as they are here considered as *included* machine operations.

Example

<pre> MACHINE MA INCLUDES MB, r2.MC PROMOTES opB1, opB3, r2.opC1, r2.opC3 ... END </pre>	<pre> MACHINE MB OPERATIONS opB1 = ... ; opB3 = END </pre>	<pre> MACHINE MC OPERATIONS opC1 = ... ; opC3 = END </pre>
--	--	--

In the example above, machine *MchA* *includes* the machines instances *MchB* and *r2.MchC*, then it promotes the operations *opB1* and *opB3* of instance *MchB* and the operations *opC1* and *opC2* of the renamed instance *r2.MchC*. The machine will then have in addition to its own operations, the four operations: *opB1*, *opB3*, *r2.opC1* and *r2.opC3*.

7.11 The EXTENDS Clause

Syntax

Clause_EXTENDS ::= "EXTENDS" ([Ident"."]Ident ["(" *Instantiating*⁺ ";" ")])⁺;

Clause_EXTENDS_B0 ::= "EXTENDS" ([Ident"."]Ident ["(" *Instantiating_B0*⁺ ";" ")])⁺;

Description

In an abstract machine or a refinement, the EXTENDS clause is equivalent to an *inclusion* (refer to section 7.9 *The INCLUDES Clause*) of the machine instances and the promotion (refer to section 7.10 *The PROMOTES Clause*) of all of the operations of the *included* machine instances.

In an implementation, the EXTENDS clause is equivalent to *importing* and promoting (refer to section 7.10 *The PROMOTES Clause*) all of the operations of the *imported* machine instances.

Restrictions

(refer to section 7.9 *The INCLUDES Clause*)

7.12 The USES Clause

Syntax

Clause_uses ::= "USES" *Ident* ⁺,"

Description

When a component *includes* a number of machines, the *included* machines may share the data of one of the *included* machines by *using* (via a USES link) the *included* machine. In the diagram below, machine *MchA* *includes* the instances of machines *MchB*, *MchC* and *MchD*. The *included* machine *MchC* is *used* (USES link) by *MchB* and *MchD*, which allows *MchB* and *MchD* to access the data in *MchC*. *MchB* and *MchD* therefore share the data in *MchC*.

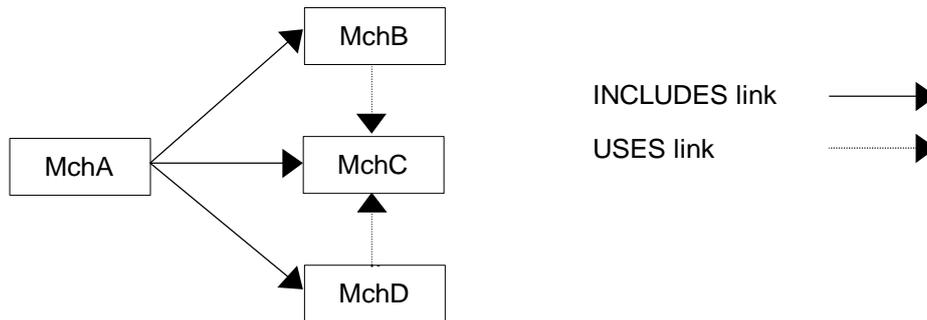


Figure 3: Principles of the USES link

Restriction

- If a machine *MA* uses an instance of machine *Mused*, then there must be, in the project, a machine that *includes* an instance of *MA* and the instance *Mused* (refer to 8.3, Rule n°9 on USES links).
- A machine that uses other machines cannot be refined. It constitutes an abstract module (refer to section 8.2 *Module B*) and it must neither be *seen* nor *imported* by other components.

Use

Let *MA* be an abstract machine, that *uses* other machines. The names in the USES list designate machine instances *used* by *MA*. An instance of *MA*, as well as the instances of machines *used* must all be *included* by a single component.

A machine that *uses* other machines must form an abstract module (refer to 8.2). It must not be *seen* or *imported* by other components.

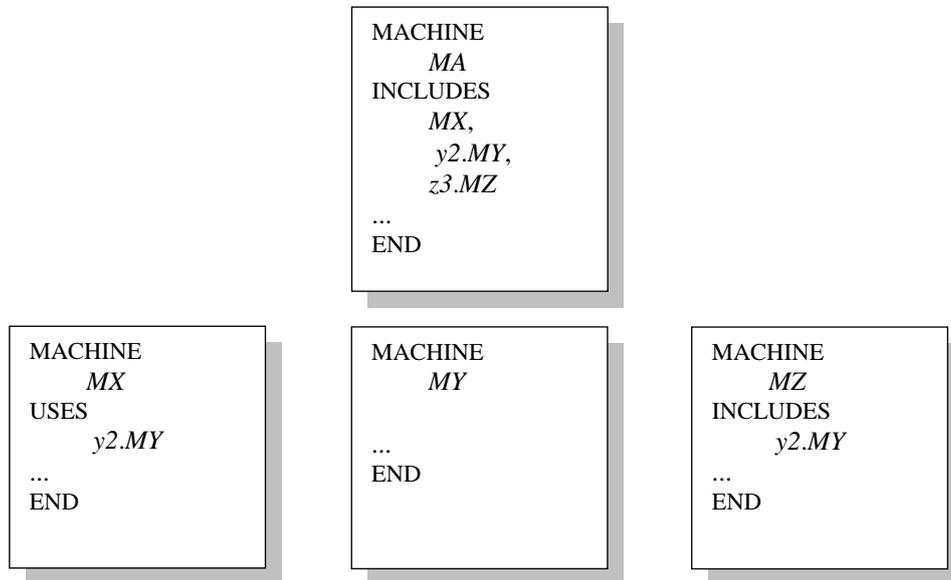
Visibility

The formal parameters of machines *used* are accessible in the PROPERTIES, INVARIANT, ASSERTIONS clauses and in the body of the initialization and the operations in the machine that *uses* it. The sets and constants are accessible in the PROPERTIES, INVARIANT, ASSERTIONS clauses and in the body of the machine initialization and the operations. The variables are accessible in the invariants and in the assertions. They are in addition, accessible in read mode in the body of the initialization and in operations. It is illegal to call the operations of a machine *used* in the initialization and in the operations of the machine.

Transitivity

The USES clause is not transitive. If a machine M_1 uses a machine M_2 that in turn uses a machine M_3 , then the formal parameters, the sets, the constants and the variables of M_3 are not accessible by M_1 .

Example



Machine MA includes the machine instances MX , $y2.MY$ and $z3.MZ$. The instance of machine $y2.MY$ is used by the machines MX and MZ (and therefore by the instances of machines MX and $z3.MZ$).

Visibility tables

Let M_A be a machine that uses a machine M_B . The visibility table and the visibility diagram below specify for each component of M_B , the modes of use that apply in the clauses of M_A .

Clauses of M_A	CONSTRAINTS	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTIONS	INITIALISATION / OPERATIONS
Formal parameters				Read	Read
Sets, enumerated set elements, concrete constants			Read	Read	Read
Abstract constants			Read	Read	Read
Concrete variables				Read	Read/non write
Abstract variables				Read	Read/non write
Operations					

7.13 The SETS Clause

Syntax

```

Clause_sets ::= "SETS" Set+
Set         ::= Ident
           | Ident "=" "{" Ident+ "}"

```

Description

The SETS clause defines the list of deferred sets and enumerated sets for a component.

Restrictions

1. The name of an enumerated set of a refinement must differ from the name of enumerated sets from abstract machines *seen* or *imported* by the refinement, except in the following case: an enumerated set belonging to an abstraction of the refinement may be identical to an enumerated set from a *seen* or *imported* machine by the refinement (the two enumerated sets must have the same name and the same ordered list of enumerated elements).
2. The name of an enumerated set of an implementation must differ from the name of enumerated sets from abstract machines *seen* or *imported* by the implementation, except in the following case: an enumerated set belonging to an abstraction of the implementation may be identical to an enumerated set from a *seen* or *imported* machine by the implementation (the two enumerated sets must have the same name and the same ordered list of enumerated elements).

Use

The deferred sets and the enumerated sets define basic scalar types having the same name as these sets (refer 3.1 *Typing foundations*). The deferred sets and the enumerated sets defined in a component are implicitly preserved during the refinement of the component, until its implementation.

- A deferred set is defined by its name. Deferred sets are used to designate objects the user does not want to define the implementation at a specification level. A deferred set is implicitly finished and non empty. It must be valued in the component implementation (refer to 7.17 *The VALUES Clause*). Finally, any deferred set is valued by a finite non empty integer interval.
- An enumerated set is defined by its name and by the ordered non empty list of its enumerated elements. The sets enumerated sets serve to describe a listing. The elements in an enumerated set are called literal enumerated items. They use the same semantics as concrete constants using the enumerated set type.

List of deferred and enumerated sets of a component

The list of deferred and enumerated sets of an abstract machine gathers the sets defined in the machine and in the machines *included* by the machine.

The list of deferred and enumerated sets of a refinement gathers, the sets defined in the refinement and those of the abstraction of the refinement.

The list of deferred and enumerated sets of an implementation gathers the sets defined in the implementation, those of the abstraction of the implementation, or finally those from machines *included* by the implementation.

Visibility

The deferred sets, the enumerated sets and the elements of an enumerated set from an abstract machine are accessible in the machine, within the INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS, INITIALISATION and OPERATIONS clauses. They are also accessible by the components that *see*, *include*, *use* or *import* the machine.

The deferred sets, the enumerated sets and elements of an enumerated set from a refinement are also accessible in the refinement, within the INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS, INITIALISATION and OPERATIONS clauses.

The deferred sets, the enumerated sets and the elements of an enumerated set in a refinement are accessible in the refinement, within the IMPORTS, EXTENDS, PROPERTIES, VALUES, INVARIANT, ASSERTIONS, INITIALISATION and OPERATIONS clauses.

Example

```
MACHINE
  MA
SETS
  POSITION ;
  MOVE = {Stop, Forward, Reverse} ;
  DIRECTION = {North, South, East, West}
...
END
```

```
IMPLEMENTATION
  MA_i
REFINES
  MA
IMPORTS
  MB
SETS
  SPEED ;
  SIGNAL = {Red, Orange, Green}
VALUES
  POSITION = 0 .. 100 ;
  SPEED = -10 .. 10
...
END
```

```
MACHINE
  MB
SETS
  MOVE = {Stop, Forward, Reverse} ;
...
END
```

In the example above, the abstract machine *MA* defines the *POSITION* deferred set and the enumerated sets *MOVE* and *DIRECTION*. *MA_i*, the implementation of *MA* defines a new *SPEED* deferred set and a new enumerated set *SIGNAL*. The two deferred sets in *MA_i* are valued in the *VALUES* clause. The *MA_i* implementation *imports* the *MB* machine that has an enumerated set *MOVE* that is identical to the one in *MA_i*, as it has the same name and the same ordered list of enumerated elements.

7.14 The CONCRETE_CONSTANTS Clause

Syntax

```
Clause_concrete_constants ::=
    "CONCRETE_CONSTANTS" Ident+","
    | "CONSTANTS" Ident+","
```

Description

The CONCRETE_CONSTANTS clause defines the list of concrete constants of a component.

A concrete constant is a data item that can be implemented in a programming language whose value remains constant and that is implicitly preserved during refinement and until implementation. A concrete constant may be an integer or a Boolean value, an element in an abstract or enumerated set, a finite and non empty interval of \mathbb{Z} or a deferred set, or an array.

Restrictions

1. The name of a new concrete constant of a refinement or of an implementation must differ from the name of the constants (concrete or abstract) in the abstraction, except in the following case: a concrete constant may refine an abstract constant of the same name of the abstraction, it is then implicitly equal to the abstract constant, and both constants must have the same type.

Use

The CONCRETE_CONSTANTS and CONSTANTS clauses are equivalent, they may therefore be used indifferently.

The concrete constants of an abstract machine gathers the concrete constants defined in the machine and thoes of the *included* machines (refer to 7.9 The INCLUDES). The concrete constants of a refinement gathers the concrete constants defined in the refinement, those of the abstraction and those of the *included* machines. The concrete constants of an implementation gathers the concrete constants defined in the implementation and those of the abstraction.

The typing and the properties of concrete constants are expressed in the PROPERTIES clause.

Each concrete constant belonging to a component must be valued in the component implementation (refer to 7.17 The VALUES).

Each concrete constant defined in an abstract machine must be typed in the machine PROPERTIES clause. A concrete constant defined in a refinement or in an implementation may be:

- A new concrete constant. It must then be explicitly typed in the PROPERTIES clause of the refinement or the implementation.
- A refinement of an abstract constant, if its name is identical to that of an abstract constant from the abstraction. To do so, the abstract constant must be of the same type as a concrete constant. It is then implicitly typed by a gluing predicate meaning that the concrete constant of the refinement equals the abstract constant of the abstraction.

Visibility

The concrete constants of an abstract machine are accessible in read mode in the PROPERTIES, INVARIANT, ASSERTIONS, INCLUDES, EXTENDS clauses and in the body of the initialization and of the operations of the machine. They are accessible in read mode via the components that *see*, *include*, *use* or *import* this machine.

The concrete constants of a refinement are accessible in read mode in the PROPERTIES, INVARIANT, ASSERTIONS, INCLUDES, EXTENDS clauses and in the body of the initialization and of the refinement operations.

The concrete constants of an implementation are accessible in read mode in the PROPERTIES, INVARIANT, ASSERTIONS, IMPORTS, EXTENDS modes and in the body of the initialization and the refinement operations. They are also accessible in write mode in the VALUES clause.

Example

<pre> MACHINE MA CONCRETE_CONSTANTS PosMin, PosMax ABSTRACT_CONSTANTS PosInit PROPERTIES PosMin ∈ INT ∧ PosMax ∈ NAT ∧ PosInit ∈ INT ... END </pre>	<pre> IMPLEMENTATION MA_i REFINES MA CONCRETE_CONSTANTS PosAverage, PosInit PROPERTIES PosAverage ∈ INT VALUES PosMin = -100 ; PosMax = 100 ; PosAverage = 0 ; PosInit = 50 ... END </pre>
---	--

The abstract machine *MA* defines two concrete constants *PosMin* and *PosMax* and an abstract constant *PosInit*. The implementation *MA_i* of *MA* defined the new concrete constants *PosAverage* and *PosInit*. The latter refines the abstract constant with the same name as *MA*. All of the constants of *MA_i* (*PosMin*, *PosMax*, *PosAverage* and *PosInit*) are valued in the implementation.

7.15 The ABSTRACT_CONSTANTS Clause

Syntax

Clause_abstract_constants ::= "ABSTRACT_CONSTANTS" Ident⁺;

Description

The ABSTRACT_CONSTANTS clause contains the list of abstract constants for an abstract machine or a refinement.

An abstract constant is a data item with a constant value that will be refined in the component refinement.

Restrictions

1. The name of the abstract constant of a refinement must be different from the name of the constants (concrete or abstract) in the abstraction, except in the following case: an abstract constant may refine a homonymous abstract constant of the abstraction, it is then implicitly equal to the abstract constant of the abstraction.

Use

The name of the clause is followed by a list of identifiers that represent the name of the abstract constants. The type and the properties of the abstract constants are expressed in the PROPERTIES clause.

Each abstract constant defined in a refinement may be:

- A new abstract constant. It must be typed and may possibly have other properties expressed in the PROPERTIES clause.
- A refinement of an abstract constant. If it has the same name as an abstract constant of the abstraction, it is then unnecessary to type the abstract constant in the PROPERTIES clause as it is typed by default using an implicit gluing property that means that the new abstract constant equals the abstract constant of the abstraction. Other properties of the constant may be expressed in the PROPERTIES clause.

If M_n refines M_{n-1} , the abstract constants of M_{n-1} may also be refined as concrete constants of M_n (refer to 7.14 *The CONCRETE_CONSTANTS Clause*). If an abstract constant of M_{n-1} is not refined as an abstract constant, nor as a concrete constant, then it is no longer a part of the constants of M_n . Then it is said to disappear in M_n .

The ABSTRACT_CONSTANTS clause is illegal in an implementation. Unlike the concrete constants, the abstract constants are not systematically implementable in a programming language.

Visibility

The abstract constants of a machine are accessible in read mode in the INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS, INITIALISATION and OPERATIONS clauses of the machine. They are accessible in read mode by the components that *import*, *see*, *include* or *use* this machine (refer to Appendix C. *Visibility Tables*).

The abstract constants of a refinement are accessible in read mode in the PROPERTIES, INVARIANT, ASSERTIONS, INCLUDES, EXTENDS, INITIALISATION and OPERATIONS clauses of the refinement.

The abstract constant of a refinement can be used in the INCLUDES, EXTENDS, PROPERTIES, INVARIANT, ASSERTIONS, INITIALISATION and OPERATIONS clauses of the refinement.

Example

```

MACHINE
  Network
ABSTRACT_CONSTANTS
  MaxNbrSubscribers,
  Connection
PROPERTIES
  MaxNbrSubscribers ∈ NAT ∧
  Connection ∈ 0 .. MaxNbrSubscribers ↔ 0 .. MaxNbrSubsc
  rbers
...
END

```

```

REFINEMENT
  Network_r
REFINES
  Network
ABSTRACT_CONSTANTS
  Connection,
  InitSubscribers
PROPERTIES
  InitSubscribers ⊂ NAT
...
END

```

The *Network* abstract machine defines two abstract constants *MaxNbrSubscribers* and *Connection*. *Network_r*, the refinement of *Network* defines the new abstract constant *InitSubscribers* and the constant *Connection*, with the same name as an abstract constant of the *Network* abstract machine. As *Connection* is implicitly equal to the constant of the abstraction, the redefinition is used to retain this abstract constant and its properties in the refinement.

7.16 The PROPERTIES Clause

Syntax

Clause_properties ::= "PROPERTIES" *Predicate*^{+"^"}

Description

The PROPERTIES clause is used to type the constants (concrete and abstract) defined in a component and to express properties for these constants.

Restrictions

1. Each concrete or abstract constant, defined in a component and not homonymous with a constant of the possible component abstraction, must be typed in the component PROPERTIES clause by a typing predicate (refer to 3.1 *Typing foundations*) located at the highest level of syntactical analysis in a list of conjunctions.
2. Each constant defined in a refinement or an implementation homonymous with a constant of the abstraction does not need to be typed as it is implicitly typed by a predicate that means that the new constant is equal to the homonymous constant of the abstraction.
3. If a constant of a *included* or *seen* machine instance by a refinement is homonymous with and has the same characteristic (abstract or concrete) as a constant of the refinement abstraction, then both constants refer to the same data and must have the same type.
4. If a constant of a machine instance *imported* or *seen* by an implementation is homonymous with and has the same characteristic (abstract or concrete) as a constant of the implementation abstraction, then both constants refer to the same data and must have the same type.

Use

The PROPERTIES clause is followed by a list of predicates separated by '^' conjunctions. The PROPERTIES clause of an abstract machine is used to type the constants of the machine and to define their properties.

The PROPERTIES clause in a refinement is used to type the new constants of the refinement and to define their properties, especially those which explicitate the gluing property between constants of the refinement and constants of the abstraction. Each new constant must be typed by a typing predicate. It is unnecessary to type a homonymous constant, as it is typed by default by an implicit gluing predicate that means that the new constant equals the abstract constant of the abstraction.

The use of the PROPERTIES clause in a refinement and in an implementation are similar. However, as the declaration of abstract constants is illegal in an implementation, only the new concrete constants of the implementation must be typed in typing predicates in the PROPERTIES clause.

Typing of constants

The constants must be typed in one of the predicates located at the highest level of syntax analysis in the PROPERTIES clause separated by '^' conjunctions, using abstract

data typing predicates for the abstract constants (refer 3.1 Typing foundations) and typing predicates for concrete constants (refer to 3.1 Typing foundations).

Properties of constants

The properties predicate for the constants allows expressing general properties that relate to constants.

Gluing Properties

The PROPERTIES clause of a refinement is used to specify the gluing property between the constants declared in the refinement and the constants of the abstraction. A predicate expressing a gluing property is called a gluing predicate. A gluing predicate may also be a typing predicate. Remember that implicit gluing predicates start the PROPERTIES clause for all homonymous gluing constants.

Visibility

In a machine PROPERTIES clause, the machine sets and constants are accessible. The sets and the constants of the *included* or *seen* machines are also accessible. The machine parameters, as well as the parameters of machines used are not accessible.

Example

```

MACHINE
  MA
  CONCRETE_CONSTANTS
    Cst1
  ABSTRACT_CONSTANTS
    Cst2
  PROPERTIES
    Cst1 ∈ INT ∧
    Cst2 ∈ ℕ ∧
    (Cst1 < 0 ⇒ Cst2 = 0)
  ...
END

```

The abstract machine *MA* defines the concrete constant *Cst1* and the abstract constant *Cst2*. These two constants are typed in the typing predicates located at the highest level of syntax analysis in a list of conjunctions. The implication that follows these typing predicates expresses a property that covers *Cst1* and *Cst2*. It should be noted that the brackets around it are required in this case, because of the priority level of '⇒' that is lower than that of '∧' (without these brackets, the predicate would be analyzed as $(Cst1 \in \text{INT} \wedge Cst2 \in \mathbb{N} \wedge Cst1 < 0) \Rightarrow Cst2 = 0$; then the typing predicates of *Cst1* and *Cst2* would no longer be in a list de conjunctions at the highest syntax level).

7.17 The VALUES Clause

Syntax

```

Clause_values ::= "VALUES" Valuation+
Valuation    ::= Ident "=" Term
              | Ident "=" "Bool" "(" Condition ")"
              | Ident "=" Expr_array
              | Ident "=" Interval_B0
Expr_array   ::= Ident
              | "{" ( Simple_term+ "→" Term )+ ";" "}"
              | ( Range+ "x" "x" "{" Term "}" )+ "v"
Interval_B0  ::= Arithmetical_expression ".." Arithmetical_expression
              | Number_set_B0
Number_set_B0 ::= "NAT"
               | "NAT1"
               | "INT"
Range        ::= Number_set_B0
               | "BOOL"
               | Interval_B0
               | Ident

```

Description

The `VALUES` clause is used to give a value to concrete constants (refer to section 7.14 *The CONCRETE_CONSTANTS Clause*) and the deferred sets (refer to section 7.13 *The SETS Clause*) of the implementation.

Restriction

1. Each concrete constant or each deferred set must be valued at most once. If it is valued once, it is explicitly valued, else it is implicitly valued.
2. A concrete constant or a deferred set implicitly valued must be homonymous with a concrete constant or a deferred set of a *seen* or *imported* machine. In the case of the valuation of a concrete constant, the two homonymous constants must have the same type.
3. If a concrete constant or a deferred set of the implementation is used in the right hand part of a valuation, it has to be previously explicitly valued or implicitly valued.

Use

The name of the `VALUES` clause is followed by a list of valuations. Each valuation is used to explicitly assign a value to a concrete constant or a deferred set. It comprises the name of the data to value, followed by the equals operator '=' and the value of the data. Explicit valuations are described as follows: valuation of scalar concrete constants, of array constants, of interval constants and of deferred sets.

Valuation of deferred sets

When valuing a deferred set *AbsSet*, the type that represents this set changes. It takes the type of the set that values it.

There are three ways to value *AbsSet*:

- by a deferred set *AbsSet2* of a *seen* or *imported* machine. The *AbsSet* type then becomes *AbsSet2*.
- implicitly, by a homonymous set of a *seen* or *imported* machine. The *AbsSet* type remains the same. In practice, as the two deferred sets have the same name, they represent the same type.
- by an integer interval, that is implementable and not empty. The *AbsSet* type becomes \mathbb{Z} .
- by a deferred set *AbsSet3* of the implementation, which has been already valued in the VALUES clause or which is valued by homonymy with a deferred set of a *seen* or *imported* machine.

It is necessary to prove that the valuation of a deferred sets is finite and not empty. If a deferred set is valued by an interval with bounds that are arithmetical expressions, it is necessary to prove that each sub-expression of these bounds belongs to INT.

The types of implementation data, built with type *AbsSet*, also change. Each occurrence of *AbsSet* is replaced by the type of the valuation expression. The scope of this change of type comprises a part of the VALUES clause, after valuing *AbsSet* as well as the IMPORTS, EXTENDS, PROPERTIES, INVARIANT, ASSERTION, INITIALISATION and OPERATIONS clauses of the implementation.

This change of type may be of interest in cases where the valuation by an interval of integers is used (any deferred set will in time be valued by an integer interval, the first two cases of valuation only defer this valuation). When the type *AbsSet* changes to type \mathbb{Z} , the data belonging to *AbsSet* may receive integer values (especially literal integers) and be handled using arithmetical operators, as these operators are only defined in B for integers. They will therefore become concretely usable.

Example

<pre> MACHINE MA SETS AbsSet CONCRETE_VARIABLES var1, var2, var3 INVARIANT var1 ∈ AbsSet ∧ var2 ∈ AbsSet ∧ var3 ∈ AbsSet ... END </pre>	<pre> IMPLEMENTATION MA_i REFINES MA VALUES AbsSet = 0 .. 100 INITIALISATION var1 := 0 ; var2 := 100 ; var3 := (var1 + var2) / 2 ... END </pre>
---	---

In the example above, concrete variables *var1*, *var2* and *var3*, defined in *MA*, are of type *AbsSet*. In the implementation *MA_i*, as *AbsSet* is valued by an integer interval, the type of variables *var1*, *var2* and *var3* becomes \mathbb{Z} . They may henceforth be initialized using arithmetical expressions.

Example of set valuation

<pre> MACHINE MA SETS Ens1 ; Ens2 ; Ens3 ; Ens4 ... END </pre>	<pre> MACHINE MB SETS Ens5, Ens6 CONCRETE_CONSTANTS c1, interv, intervA PROPERTIES c1 ∈ 0 .. 100 ∧ interv ⊆ NAT ∧ intervA ⊆ Ens6 END </pre>
<pre> IMPLEMENTATION MA_i REFINES MA SEES MB VALUES Ens1 = 0 .. 2 × c1 + 1 ; Ens2 = Interv ; Ens3 = IntervA ; Ens4 = Ens6 ... END </pre>	

In the example above, the deferred set *Ens1* is valued by an integer interval. *Ens2* is valued by an integer interval constant of *seen* machine *MB*. *Ens3* is valued by an deferred type constant interval from *MB*. *Ens4* is valued by deferred set *Ens6* from *MB*. Finally, *Ens5* is implicitly valued by the homonymous deferred set of *MB*.

Valuation of scalar concrete constants

Each scalar concrete constant is valued according to its type. If it belongs to \mathbb{Z} , it must be valued by an arithmetical expression. It is then necessary to prove that each arithmetical sub-expression is in fact defined and that it is contained in the set of implementable integers INT (refer to section 3.1, *Typing foundations*). If the concrete constant belongs to BOOL, it must be valued by a Boolean expression. If it is an enumerated or deferred type, it must be valued by an enumerated or deferred type constant.

In the case of deferred type constants, where the deferred set has already been valued in the VALUES clause, remember that the type of the constant is changed; it refers to the type of the set that values the deferred set.

Example

In the example below, concrete constant *c1* is valued by a literal integer. Constant *c2* is valued by an arithmetical expression, using concrete constant *CstB2* of *seen* machine *MB*. Constant *c3* is valued by an enumerated element from the enumerated set *COLOR* declared in *MB*. Constant *c4* is valued legitimately by a literal integer as it is an integer type since the valuation of *EnsAbs1* by the integer interval 0 .. 12. Finally, constant *c5* is valued by concrete constant *cteB1* of *MB*.

```

MACHINE
  MA
SEES
  MB
SETS
  EnsAbs1
CONCRETE_CONSTANTS
  c1, c2, c3, c4, c5
PROPERTIES
  c1 ∈ INT ∧
  c2 ∈ NAT ∧
  c3 ∈ COUL ∧
  c4 ∈ EnsAbs1 ∧
  c5 ∈ EnsAbs2
...
END

```

```

MACHINE
  MB
SETS
  COLOR = { Red, Green, Blue };
  EnsAbs2
CONCRETE_CONSTANTS
  cteB1,
  cteB2
PROPERTIES
  cteB1 ∈ EnsAbs2 ∧
  cteB2 ∈ - 10 .. 10
END

```

```

IMPLEMENTATION
  MA_i
REFINES
  MA
SEES
  MB
VALUES
  c1 = - 100 ;
  c2 = cteB22 - 4 ;
  c3 = Blue ;
  EnsAbs1 = 0 .. 512 ;
  c4 = 42 ;
  c5 = cteB1
...
END

```

Valuation of array concrete constants

An array concrete constant may be valued in three different ways:

- by an array concrete constant from a *seen* or *imported* machine.
- by a set of maplets. A maplet is used to represent an n -uplet in which the $n-1$ first elements designate the indices of the array element and in which the last element refers to the value of the array element. The indices of a maplet must be scalar literal values, while the value of a maplet must be a scalar value.
- by an array where all elements have the same value. This is done using the cartesian product between the domain of the array and a singleton containing the value of all elements of the array.

In the valuation of an array whose elements are integers, if an array value is defined by an arithmetical expression, it is necessary to prove that each sub-expression belongs to INT (refer to section 3.1 Typing foundations).

Example

In the example below, the constant $t1$ is valued using the constant $t4$ in the machine *seen MB*. The constant $t2$ is valued by a set of maplets: the element with the index (1) takes the value FALSE, (2) takes the value TRUE. The constant $t3$ is valued by the array $EnsAbs \times \text{BOOL} \times \{0\}$ which associates to any element of the array domain, the value 0.

```

MACHINE
  MA
SEES
  MB
CONCRETE_CONSTANTS
  t1, t2, b3
PROPERTIES
  t1 ∈ (0 .. 2) × BOOL → EnsAbs ∧
  t2 ∈ (1 .. 2) →>> BOOL ∧
  t3 ∈ EnsAbs × BOOL → INT
...
END

```

```

MACHINE
  MB
SETS
  EnsAbs
CONCRETE_CONSTANTS
  t4
PROPERTIES
  t4 ∈ (0 .. 2) × BOOL → EnsAbs
...
END

```

```

IMPLEMENTATION
  MA_i
REFINES
  MA
SEES
  MB
VALUES
  t1 = t4 ;
  t2 = { 1 ↦ FALSE, 2 ↦ TRUE } ;
  t3 = EnsAbs × {BOOL} × {0}
...
END

```

Valuation of concrete constants intervals

Each concrete constant interval is valued according to its type. If it is an integer type, it may be valued by an integer constant interval or by an interval with bounds that are arithmetical expressions. If they are a deferred type, they must be valued by a deferred type constant interval.

In the valuation of an integer interval using arithmetical expressions, it is necessary to prove that each sub-expression belongs to INT.

Example

In the example below, constant interval $c1$ is valued using constant $cteInterv1$ of *seen machine MB*. Constant interval $c2$ is valued by the integer interval $0..MAXINT/2-1$, where the upper bound is an arithmetical expression. Deferred set $EnsAbs1$ is valued by the integer interval $0..100$. As a result, the data with type $EnsAbs1$ becomes \mathbb{Z} . The constant interval $c3$ belonging to $EnsAbs1$ is valued by the interval $1..6$. The constant interval $c4$ belonging to the deferred set $EnsAbs2$ is valued by the constant interval $cteInterv2$ of *seen machine MB*, this latter constant belonging to $EnsAbs2$.

```

MACHINE
  MA
SEES
  MB
SETS
  EnsAbs1
CONCRETE_CONSTANTS
  c1, c2, c3, c4
PROPERTIES
  c1 ⊆ INT ∧
  c2 = 0 .. (MAXINT / 2 - 1) ∧
  c3 ⊆ EnsAbs1 ∧
  c4 ⊆ EnsAbs2
...
END

```

```

IMPLEMENTATION
  MA_i
REFINES
  MA
SEES
  MB
VALUES
  c1 = cteInterv1 ;
  c2 = 0 .. (MAXINT / 2 - 1) ;
  EnsAbs1 = 0 .. 100 ;
  c3 = 1 .. 6 ;
  c4 = cteInterv2
...
END

```

```

MACHINE
  MB
SETS
  EnsAbs2
CONCRETE_CONSTANTS
  cteInterv1, cteInterv2
PROPERTIES
  cteInterv1 ⊆ INT ∧
  cteInterv2 ⊆ EnsAbs2
...
END

```

Visibility

In the VALUES clause of an implementation, the concrete constants and the enumerated sets of *seen* and *imported* machines are accessible in read mode. The concrete constants and the deferred sets of the implementation can only appear in the right hand part of valuations, after they have been valuated.

7.18 The CONCRETE_VARIABLES Clause

Syntax

Clause_concrete_variables ::= "CONCRETE_VARIABLES" (Ident^{+"})^{+"}

Description

The CONCRETE_VARIABLES clause defines the concrete variables of a component.

A concrete variable is a data item that can be implemented in a programming language, a variable that therefore does not need to be refined as it is implicitly preserved during refinement and through the implementation. This property then allows access in direct read mode to the concrete variables of a machine in the implementations that will *import* this machine, without necessarily requiring the use of a read service as is necessarily the case for abstract variables.

Restrictions

1. The concrete variables declared in a machine must not be renamed.
2. The name of a new concrete variable of a refinement or an implementation must differ from the name of the variables (concrete or abstract) in the abstraction, except in the following case: a concrete variable may refine a homonymous abstract variable of the abstraction, it is then implicitly equal to the abstract variable.

Use

The concrete variables of an abstract machine gather the concrete variables defined in the machine and those of the *included* abstract machines (refer to section 7.9 *The INCLUDES Clause*). The concrete variables of a refinement gather the concrete variables defined in the refinement, those of the abstraction and those of the *included* abstract machines. The concrete variables of an implementation gather the concrete variables defined in the implementation and those of the abstraction.

Typing and other invariant properties of concrete variables are expressed in the INVARIANT clause.

Each concrete variable of a component must be initialized in the component INITIALISATION clause (refer to 7.22 *The INITIALISATION Clause*).

Each concrete variable defined in an abstract machine must be typed in the machine INITIALISATION clause. A concrete variable defined in a refinement or in an implementation may be:

- a new concrete variable. It must then be explicitly typed in the INITIALISATION clause of the refinement or the implementation.
- a refinement of an abstract variable, if it has the same name as an abstract variable in the abstraction. In this case, the abstract variable must have the same type as the concrete variable (scalar or array type). It is then implicitly typed by a gluing predicate that means that the concrete variable is equal to the homonymous abstract variable.

Visibility

The concrete variables of a component can be used in the INVARIANT, ASSERTIONS clauses of this component and in its refinements. They are accessible in read and in write modes in the body of the initialization and of the operations of the component. The concrete variables declared in a machine are accessible in read-only by the components that *see*, *include*, *use* or *import* this machine.

The concrete variables of a refinement or an implementation are accessible in the INVARIANT, ASSERTIONS clauses of this refinement and are accessible in read and in write modes in the body of the initialization and the operations of the machine.

Example

```

MACHINE
  MA
  CONCRETE_VARIABLES
    Prod,
    MonthProd
  ABSTRACT_VARIABLES
    ExternalProd
  INVARIANT
    Prod ∈ INT ∧
    MonthProd ∈ ( 1 .. 12 ) → INT ∧
    ExternalProd ∈ INT
  INITIALISATION
    Prod := 0 ;
    MonthProd := ( 1 .. 12 ) → INT ;
    ExternalProd := 0 ;
  ...
END

```

```

IMPLEMENTATION
  MA_i
  REFINES
    MA
  CONCRETE_VARIABLES
    Receipts,
    Charges,
    ExternalProd
  INVARIANT
    Receipts ∈ INT ∧
    Charges ∈ INT
  INITIALISATION
    Prod := 0 ;
    MonthProd := ( 1 .. 12 ) × {0} ;
    ExternalProd := 0 ;
    Receipts := 0 ;
    Charges := 0 ;
  ...
END

```

The abstract machine *MA* defined two concrete variables *Prod* and *MonthProd* and an abstract variable *ExternalProd*. The *Prod* and *ExternalProd* variables are implementable integers and the *MonthProd* variable is an array of implementable integers with an index in the interval 1..12. All of these variables are initialized in the INITIALISATION clause. The *MA_i* implementation of *MA* defines the new concrete variables *Receipts*, *Charges* and *ExternalProd*. The latter refines the abstract constant with the same name as *MA*. All of the variables of *MA_i* (*Prod*, *MonthProd*, *ExternalProd*, *Receipts* and *Charges*) are initialized in the INITIALISATION clause.

7.19 The ABSTRACT_VARIABLES Clause

Syntax

```
Clause_abstract_variables ::=
    "ABSTRACT_VARIABLES" Ident*,"
|
    "VARIABLES" Ident*,"
```

Description

The ABSTRACT_VARIABLES clause defines new abstract variables into a machine or a refinement. An abstract variable is a data item of any type, which may be refined during component refinement.

Restrictions

1. The abstract variables declared in a machine must not be renamed.
2. The name of a new abstract variable of a refinement or an implementation must be different from the name of the variables (concrete or abstract) of the abstraction, except in the following case: an abstract variable may refine a homonymous abstract of the abstraction, it is then implicitly equal to the abstract variable.

Use

The name of the clause is followed by a list of identifiers that represent the names of abstract variables. The abstract variables must be typed (refer to section 3.1 Typing foundations) and may have other invariant properties in the INVARIANT clause. They must be initialized in the INITIALISATION clause.

In a refinement, each abstract variable defined may be:

- A new abstract variable. Its name must then be an identifier that is not renamed. The abstract variable must be typed and may have other invariant properties in the INVARIANT clause.
- The refinement of an abstract variable, if it has the same name as an abstract variable of the refined component. The identifier of the variable is renamed if the variable of the refined component comes from a renamed *included* machine (or one that is transitively *included*) and that is renamed. In this case the abstract variable is implicitly typed by a gluing invariant meaning that the new abstract variable is equal to the homonymous abstract variable of the refined component. Other invariant properties about the abstract variable may also be expressed in the INVARIANT clause.

If refinement M_n refines M_{n-1} , the abstract variables of M_{n-1} may also be refined as concrete variables of M_n . If a variable of M_{n-1} is not refined as an abstract variable, nor as a concrete variable, then it disappears in M_n . It is therefore no longer a part of the variables of component M_n .

The refinement abstract variables that do not come from an *included* machine instance by the refinement must be initialized in the INITIALISATION clause.

In an implementation, it is illegal to define abstract variables.

Visibility

The abstract variables of a component are accessible in the INVARIANT and ASSERTIONS clauses of this component. They are accessible in read and in write mode in the body of

the initialization and the operations of the machine and any refinement. They are all accessible in read mode through the components that *import*, *see*, *include* or *use* this machine (refer to Appendix C. *Visibility Tables*).

Let M_A and M_B be machines. If M_A *sees* M_B , the abstract variables of M_B are accessible in M_A in read mode in the body of the initialization and in operations in M_A . If M_A *uses*, or *includes* M_B , the abstract variables of M_B are accessible in M_A in the INVARIANT and ASSERTIONS clauses and they are accessible in read mode in the body of the initialization and of the operations.

The abstract variables of a refinement are accessible in the INVARIANT, ASSERTIONS clauses of this refinement and are accessible in read and write modes in the body of the initialization and in the operations of the machine.

If M_n refines M_{n-1} , the abstract variables of M_{n-1} , that disappear in M_n are only accessible in M_n , in the INVARIANT and ASSERTIONS clauses as well as in the INITIALISATION and OPERATIONS clauses, in predicates used for substitutions/assertion and for loop variants and invariants. They are no longer accessible in the refinements of M_n .

Let M_A be a refinement and M_B a machine. If M_A *sees* M_B , the abstract variables of M_B are accessible in M_A in read mode in the body of the initialization and the operations.

If M_A *includes* M_B , the abstract variables of M_B are accessible in M_A in the INVARIANT and ASSERTIONS clauses and they are accessible in read mode in the body of the initialization and the operations.

The abstract variables declared in a refinement are not accessible by components external to the module.

7.20 The INVARIANT Clause

Syntax

Clause_invariant ::= "INVARIANT" *Predicate*^{+"^"}

Description

The INVARIANT clause contains, within a predicate called invariant, the typing of variables declared in the component and also the properties of these variables.

The invariant expresses the invariant properties of variables of the abstract machine. It is necessary to prove that component initialization (refer to section 7.22 *The INITIALISATION Clause*) established the invariant and that each time a machine operation is called, the invariant is preserved. External components may access the concrete variables of an abstract machine in read mode, but not in write mode in order to avoid breaking the invariant.

The invariant of a refinement or an implementation is used to express the linkage between the new variables of the component and the variables of the abstraction called gluing invariant.

Restrictions

1. Each concrete or abstract variable, defined in a component non homonymous to a variable of a possible component abstraction, must be typed in the INVARIANT clause of the component using a typing predicate (refer to section 3.1 *Typing foundations*) located at the highest syntactical level in a list of conjunctions.
2. Each variable defined in a refinement or an implementation homonymous to a variable of the abstraction must not be typed, as it is implicitly typed by a predicate meaning that the new variable is equal to the variable homonymous variable of the abstraction.
3. If a variable of a machine instance *included* by a refinement has the same name and the same characteristic (abstract or concrete) as the variable of the refinement abstraction, then both variables designate the same data and must be of the same type.
4. If a variable of a machine instance *imported* by an implementation has the same name and the same characteristic (abstract or concrete) as the variable of the implementation abstraction, then the both homonymous variables designate the same data and must be of the same type.

Use

The name of the INVARIANT clause is followed by a list of predicates used to type the variables defined in the machine and to define their invariant properties.

The invariant of a refinement is used to type the new variables of the refinement and to define its properties, especially the linkage between the refinement variables and those of the abstraction.

The invariant of an implementation is similar to that of a refinement. It defines the gluing invariant between the implementation variables and the variables of instances of machines *imported* by the implementation.

Typing of variables

The variables must be typed in one of the predicates located at the highest level of syntactical analysis in the INVARIANT clause separated by '^' conjunctions and using the abstract data typing predicates for the abstract variables (refer to section 3.1 Typing foundations) and the concrete variables typing predicate for the concrete variables (refer 3.1 Typing foundations).

The variables declared in the refinement may be split into two groups: the new variables and the variables that refine homonymous abstract variables of the abstraction. Each new variable must be typed by a typing invariant. A variable that refines a homonymous abstract variable of the abstraction, is implicitly typed by a gluing invariant that means that the new variable is equal to the homonymous abstract variable of the abstraction.

As the new variables declared in an implementation may only be concrete variables, the typing of variables in the invariant only refer to concrete variables declared in the implementation.

Linkage between variables

The invariant of a refinement is used to specify the linkage between the variables declared in the refinement and the variables from its abstraction. Each predicate in the invariant that defines this kind of linkage is called a gluing invariant. A gluing invariant may take the form of a typing predicate or a property. Remember also that an implicit gluing invariant by default links two homonymous variables, one being declared in the refinement, the other in its abstraction.

The abstract variables of implementation abstraction may be linked to the concrete or abstract variables of instances of *imported* machine (refer to the example below).

Let M_i be an implementation. If a concrete variable of the abstraction of M_i has the same name as a concrete variable of an *imported* machine instance $Mimp$, then the two variables are automatically linked by an implicit gluing invariant that means that the variables are equal. As a result, the two variables must be of the same type. The two variables will merge; it is said that the variable of M_i is implemented on the homonymous variable of $Mimp$. The concrete variable of M_i then acts as a reference on the variable with the same name in $Mimp$. The instance of machine $Mimp$ becomes responsible for managing the concrete variable and especially its initialization.

It is possible to implement a concrete variable from the M_i implementation using a concrete variable with a different name from an *imported* machine instance $Mimp$ by writing in the invariant, the equality between two variables. However, in practice this case is not interesting, since the two variables do not merge. To prove the invariant preservation, it is therefore necessary to modify the two variables together.

Example

In the example below, the concrete variable $var1$ from machine MA is implemented by variable $vimp1$, the abstract variable $var2$ is implicitly implemented on the homonymous variable $var2$ of the imported machine $Mimp$, and the concrete variables $var3$ and $var4$ are implemented locally as specific variables of the component. The concrete variable $var5$ is implemented on the homonymous variable $var5$ of *imported* machine $Mimp$.

```

MACHINE
  MA
ABSTRACT_VARIABLES
  var1,
  var2
CONCRETE_VARIABLES
  var5
INVARIANT
  var1 ∈ NAT ∧
  var2 ∈ BOOL ∧
  var5 ∈ INT ∧
  (var1 > var5 ∧ var2 = TRUE)
...
END

```

```

IMPLEMENTATION
  MA_i
REFINES
  MA
IMPORTS
  Mimp
CONCRETE_VARIABLES
  var3
INVARIANT
  var3 ∈ NAT ∧
  var1 = vimp /* link invariant */
  /* implicit link invariant var5 = var5 */
...
END

```

```

MACHINE
  Mimp
ABSTRACT_VARIABLES
  vimp,
  var2
CONCRETE_VARIABLES
  var5
INVARIANT
  vimp ∈ 1 .. 100 ∧
  var2 ∈ BOOL ∧
  var5 ∈ INT
...
END

```

Visibility

In a machine `INVARIANT` clause, the formal parameters, the sets (deferred and enumerated), the constants and the variables of the machine are accessible. The sets, the constants and the variables of *included* machines are accessible. The parameters, the sets, the constants and the variables of machines *used* are accessible. The sets and the constants of machines *seen* are accessible.

In the `INVARIANT` clause of a refinement, the formal parameters, the sets (abstracts and listed), the constants and the variables of the refinement are accessible. The constants and the variables of the abstraction that disappear in the refinement are accessible. The sets, the constants and the variables of *included* machines are accessible. The sets and the constants of *seen* machines are accessible.

In the invariant of an implementation, the following components are accessible:

- The formal parameters, the sets, the enumerated elements, the concrete constants and the concrete variables of the implementation,
- The abstract constants and the abstract variables of the abstraction of the implementation,
- The sets, the enumerated elements, the constants and the variables of instances of machines that are *seen* or *imported* by the implementation.

Example

```
MACHINE
  MA
  CONCRETE_VARIABLES
    var1
  ABSTRACT_VARIABLES
    var2
  INVARIANT
    var1 ∈ INT ∧
    var2 ∈ ℕ ∧
    (var1 > 0 ∧ var1 + var2 = 0)
  INITIALISATION
    var1 := MININT .. 0 ||
    var2 := - var1
  ...
END
```

7.21 The ASSERTIONS Clause

Syntax

Clause_assertions ::= "ASSERTIONS" *Predicate*^{+", "}

Description

The ASSERTIONS clause comprises a list of predicates called assertions referring to the component variables. Assertions are intermediate results deduced from the component invariant use to make the proof of the component easier.

An assertion is a lemma that must be proven from the component invariant and from the lemmas separated by ‘;’ that precede it in the ASSERTIONS clause. The order of assertions is therefore significant. In the proof obligations relating to component operations, the assertions are added as assumptions in addition to the invariant.

Example

In the example below, the concrete variable *var* is an implementable integer that verifies $var^2 = 1$. An assertion is added to indicate that the variable *var* is equal to 1 or to -1. In practice, this assertion may be proven by taking as assumption the invariant. When building other Proof Obligations for the machine, each time the invariant appears as an assumption, then this assertion will be added to make the demonstration of the Proof Obligation easier.

```
MACHINE
  MA
  CONCRETE_VARIABLES
    var
  INVARIANT
    var ∈ INT ∧
    var2 = 1
  ASSERTIONS
    var = 1 ∨ var = - 1
  ...
  END
```

7.22 The INITIALISATION clause

Syntax

```
Clause_initialization ::= "INITIALISATION" Substitution
Clause_initialization_B0 ::= "INITIALISATION" Instruction
```

Description

The INITIALISATION clause is used to initialize all of the variables of the component. It is necessary to prove that the initialization of a component establishes the invariant.

The INITIALISATION clause may be considered as the declaration of a specific operation. This operation does not have any parameters. Its role is to initialize the module variables so that they establish the component's invariant. When running the B0 code of a project, all of the module initialization operations are called in a correct dependency order, before calling the project entry point.

Restriction

1. Each component variable, which has not the same name as an instance variable of the *included* or *imported* machine, must be initialized in the INITIALISATION clause.

Use

The clause name is followed by a specification substitution. Substitutions are described in chapter *Substitutions*.

All of the component variables must be initialized during the initialization phase. The variables of machines *included* by the component must not be initialized by the component as they are already initialized in the INITIALISATION clause of their machine. On initialization of component variables, the variables of dependent machines (*included*, *used* or *seen*) are considered as previously initialized. The initialization allows modifying the variables of *included* machines by calling operations of their *included* machines. This possibility may especially be used in order to establish the invariants that express gluing properties for variables of *included* machines.

In a refinement, the initialization substitutions must be refinement substitutions. Like in a machine, all of the refinement variables that do not come from an *included* machine instance must be initialized. The variables concerned here are concrete variables of the refinement abstraction and of new variables declared in the refinement.

In an implementation, the concrete variables are initialized in the INITIALISATION clause.

The substitutions used in the initialization must however be implementation substitutions also called instructions (refer to section 7.25.4, *Instructions*). All of the implementation variables must be initialized. There are two ways to initialize a concrete variable for an implementation M_i :

- Directly, by explicitly giving a value to the concrete variable in an initialization instruction. The concrete variable is then localized in M_i .
- Indirectly, if the concrete variable has the same name as a concrete variable of an *imported* machine instance $Mimp$. Then the two homonymous variables must have the same type. The variable of M_i must not be initialized in the INITIALISATION clause of M_i since it is implemented by $Mimp$. The concrete variable of M_i then

only represents a reference to the homonymous variable of *Mimp*. As a result, the effective initialization of the concrete variable is done in *Mimp*.

The values of concrete and abstract variables of *imported* machines instances can be modified by the initialization. Remember that on machine initialization, the variables of instances of machines that are *seen* or *imported* by M_i are assumed to have already been initialized. It does however remain possible to modify the values of variables *imported* using the operations of these machines.

Visibility

Machine variables are accessible in read and write modes in the machine INITIALISATION clause. The machine parameters, sets and constants are accessible in read mode. The sets, constants and variables of instances of machines that are *included*, *used* or *seen* are accessible in read mode, and *used* machines parameters are accessible in read mode.

In the initialization of a machine M_i , it is possible to access the operations of machines *included* by M_i and accessing operations on machines *seen* by M_i . It is however illegal to access the specific operations of M_i and the operations on machines *used* by M_i .

The variables of the refinement are accessible in write mode in the component INITIALISATION clause. The parameters, sets and constants of the refinement are accessible in read mode in the initialization. The sets, constants and variables of machines *included* or *seen* by the refinement are accessible in read mode in the initialization.

In the initialization of refinement M_i , it is possible to access the operations of machines *included* by M_i and to access operations on machines *seen* by M_i . It is however illegal to access the operations of the OPERATIONS clause M_i .

In an implementation, variables are accessible in read and in write mode in the instructions of the INITIALISATION clause. They must be written before they are read. Formal parameters, enumerated elements and constants of the implementation are accessible in read mode in the initialization. The enumerated elements, the concrete constants and the concrete variables of machines *seen* or *imported* by the implementation are accessible in read mode in the initialization. The formal parameters of the implementation, the sets, constants and variables of the implementation and of instances of machines that are *seen* or *imported* are accessible in the loops invariant and variants and in the ASSERT predicates of the initialization.

Example

In the example below, abstract variable *var1* in machine *MA* is implemented by variable *vimp*, variable *var2* is implicitly implemented on homonymous variable *var2* of the *imported* machine *Mimp*, and concrete variables *var3* and *var4* are locally implemented.

```

IMPLEMENTATION
  MA_i
REFINES
  MA
IMPORTS
  Mimp
CONCRETE_VARIABLES
  var3
INVARIANT
  var3 ∈ NAT ∧
  var1 = vimp /* gluing invariant */
INITIALISATION
  var3 := 0 ;
  setv2 ( TRUE ) ;
  setvimp1 (10) ;
  var4 := - 12
  /* implicit implementation of var2 */
...
END

```

```

MACHINE
  MA
ABSTRACT_VARIABLES
  var1,
  var2
CONCRETE_VARIABLES
  var4
INVARIANT
  var1 ∈ NAT ∧
  var2 ∈ BOOL ∧
  var4 ∈ INT ∧
...
END

```

```

MACHINE
  Mimp
CONCRETE_VARIABLES
  vimp,
  var2
INVARIANT
  vimp ∈ 1 .. 100 ∧
  var2 ∈ BOOL
INITIALISATION
  vimp := 1 .. 100 ||
  var2 := BOOL
OPERATIONS
  setv2 (b0) =
  PRE
    b0 ∈ BOOL
  THEN
    var2 := b0
  END ;

  setvimp (in) =
  PRE
    b0 ∈ 1 .. 100
  THEN
    vimp := in
  END
END

```

7.23 The OPERATIONS Clause

Syntax

```

Clause_operations ::= "OPERATIONS" Operation+;
Operation         ::= Header_operation "=" Level1_substitution
Header_operation  ::= [ Ident+; "<" ] Ident+; [ "(" Ident+; ")" ]
Clause_operations_B0 ::= "OPERATIONS" Operation_B0+;
Operation_B0     ::= Header_operation "=" Level1_instruction
Substitution_body_operation ::=
    Block_substitution
    | Identity_substitution
    | Becomes_equal_substitution
    | Precondition_substitution
    | Assertion_substitution
    | Substitution_limited_choice
    | If_substitution
    | Select_substitution
    | Case_substitution
    | Any_substitution
    | Let_substitution
    | Becomes_elt_substitution
    | Becomes_such_that_substitution
    | Var_substitution
    | Substitution_call
Instruction_body_operation ::=
    Block_instruction
    | Identity_instruction
    | Becomes_equal_instruction
    | Assert_instruction
    | If_instruction
    | Case_instruction
    | Instruction_as_long_as
    | Var_instruction
    | Instruction_call

```

Restrictions

1. The formal input and output parameters of an operation must be two by two distinct.
2. In an abstract machine, the operations declared in the OPERATIONS clause must not be renamed.
3. In an abstract machine, operation input parameters must be typed in the predicate of a precondition substitution at the beginning of the operation body by typing predicates (refer to section 3.7, *Typing operation input parameters* and section 3.3, *Typing abstract data*) located at the highest level of syntactical analysis in a list of conjunction. These input parameters cannot be used in the predicate of the substitution precondition before being typed.
4. In an abstract machine, operation output parameters must be typed in the operation body by the typing substitutions (refer to section 3.9, *Typing local variables and operation output parameters*). These output parameters cannot be used in the operation body before being typed.

5. In an refinement, new operations cannot be defined.
6. In an implementation, each abstract machine operation must be declared in the OPERATIONS clause or must be promoted (refer to section 7.10, *The PROMOTES Clause*)
7. Each operation of a refinement or of an implementation must have the same name and the same formal parameters as an operation in its abstraction (the homonymous operation of the LOCAL_OPERATION clause in the case of a local operation, else the homonymous operation of the abstract machine).
8. In the OPERATIONS clause of an implementation, only the operations specified in the abstract machine of the implementation or in the LOCAL_OPERATIONS clause can be defined.
9. In an implementation, each local operation specified in the LOCAL_OPERATIONS clause must be implemented in the OPERATIONS clause.
10. The graph of the local operation call, taking into account only the implementations of the local operations of the OPERATIONS clause, must not contain cycle.

Description

The OPERATIONS clause is used to declare operations in a component. Operations form the dynamic part of B language as they may modify the values of variables. An operation may have both input and output parameters. The operations of abstract machines form the specifications of the operation for the module. The operations of a machine may be used by other machines matching of operation calls (refer to section 6.16, *Operation Call Substitution*).

The OPERATIONS clause allows to declare the implementation of local operations, specified in the LOCAL_OPERATIONS clause (refer to section 7.24, *The LOCAL_OPERATIONS Clause*).

It is necessary to prove that the operations in an abstract machine preserve the machine invariant. The operations must be refined until their implementation to become programming operations. It is necessary to prove that at each step, the operation is consistent with the operation that it refines.

Use in an abstract machine

The operations of an abstract machine comprise promoted operations (refer to section 7.10 *The PROMOTES Clause* and section 7.11 *The EXTENDS Clause*) and operations from the OPERATIONS clause.

The OPERATIONS clause is used to declare the services offered by a machine and to specify their behavior. The operations of an abstract machine constitute the dynamic part of the machine, by opposition to the data (machine sets, constants, variables and parameters) that constitute the static part. In practice, they allow the modification of data in the machine. It is necessary to prove that calling a machine operation preserves the machine invariant.

Operation parameters of developed modules (refer to section 8.2) and of base machines must be implementable as they will be associated with a code, when operations parameters of abstract modules may be of any type as they are not associated with code.

An operation comprises a header and a body.

Operation header

The operation header is made up of an identifier that designated the name of the operation and of any formal input and output parameters for the operation. The name of an operation declared in an abstract machine must not include renaming. The input parameters are represented by a list of bracketed identifiers that follow the name of the operation. The output parameters are represented by a list of identifiers that precede the name of the operation. The input and output parameters of an operation must be distinct two by two.

Passing parameters by value

In B, when an operation is called, the actual parameters are passed by copy.

- The input parameters of the operation are used to parameterize operation calls. When an operation is called, the value of each effective input parameter is copied to the corresponding formal parameter.
- The output parameters from the operation are used to pass the results of an operation call matching of values. After an operation call, the value of each formal output parameter is copied to the corresponding effective parameter.

Scopes

The scope of the formal parameters defined in an operation header is the body of the operation. The formal operation entry parameters are accessible in read-only mode in read mode only, in the substitutions. The formal operation output parameters are accessible in the substitutions in read and write mode. An output parameter has to be initialized before it can be read.

Typing rules

The operation parameters of modules with an associated code (developed modules or base machines) must be implementable types. The types allowed are those of a concrete variable (integer, Boolean, deferred set, enumerated set or array types). In the case of operation input parameters, the characters string type is also allowed, giving the ability to send a message using an operation call.

The abstract module operation parameters may be of any type (refer 3.1 *Typing foundations*).

Operation input parameters

The formal input parameters must be typed in the body of the operation, in a typing predicate. To be able to use a formal input parameter in the operation, it is necessary to type it in the text that precedes its use. An operation that has input parameters is written using a precondition substitution, that types the formal input parameters then possibly expresses other properties of these input parameters. When the operation is specified, the assumption is made that the formal input parameters verify the precondition and when this operation is called, it is necessary to prove that the effective parameters verify the precondition. The formal input parameter cannot be modified in the body of the operation. They will therefore always verify the precondition in the body of the operation.

Example

```

MACHINE
  MA
OPERATIONS
  Service1 (x1, b1, tab1, mess) =
  PRE
    x1 ∈ NAT ∧
    b1 ∈ BOOL ∧
    tab1 ∈ (0 .. 10) × (0 .. 10) ↦ INT ∧
    mess ∈ STRING ∧
    ...
  THEN
    ...
  END
END

```

Operation output parameters

The formal output parameters must be typed in the body of the operation. To be able to use a formal output parameter in the operation, it must have been typed in the text that precedes its use. The usual way of writing an operation with output parameters comprises typing them and giving them a value in “becomes equal”, “becomes part of”, “becomes such that” substitution or as effective output parameters of an operation call.

Example

```

MACHINE
  MA
OPERATIONS
  ok, res1, tab2 ← Service2 =
  BEGIN
    res1 : (res1 ∈ 0 .. 10 ∧ res1 / 2 = 0) ||
    tab2 := (0 .. 10) × (0 .. 10) → INT ||
    ...
    ok := bool ( ... )
  THEN
    ...
  END
END

```

Operation body

The body of an operation is a substitution. Only specification level substitutions are allowed (refer to chapter 6 *Substitutions*).

Example

```

MACHINE
  MA
OPERATIONS
  res_min, res_max, equal ← Compare (x1, x2)
=
  PRE
    x1 ∈ INT ∧
    x2 ∈ INT
  THEN
    res_min := min ({x1, x2}) ||
    res_max := max ({x1, x2}) ||
    egal := Bool (x1 = x2)
  END
END

```

Visibility

The component variables are accessible in read and write mode in the OPERATIONS clause of the component. The parameters, sets and constants of the component are accessible in read mode. The sets, constants and variables of *included*, *used* or *seen* machines are accessible in read mode, and the parameters of *used* machines are accessible in read mode.

In the body of an operation, it is possible to access operations on *included* machines and to access operations on *seen* machines. However, it is illegal to access operations declared in the OPERATIONS clause of the machine and operations on *used* machines.

Use in a refinement

Refining an operation

In the successive refinements of a machine, each operation of the machine, whether first declared in the OPERATIONS clause or promoted, must be refined by an operation. It is illegal to declare new operations in a refinement.

The name of each operation of a refinement must correspond to the name of an operation in the corresponding abstraction. This operation may be declared in the OPERATIONS clause or may be a operation promoted, independently of the choice made previously. In this way, each operation may be refined by:

- A specific operation in the OPERATIONS clause, whose name is that of the operation declared in the abstraction. If the name of the abstraction operation comprises a prefix, then the prefix should remain.
- An operation promoted by the refinement, the name of which is the name of the operation declared in the abstraction. If the operation promoted by the refinement comes from a machine instance *included* but not renamed, then this machine must have an operation of the same name. If the operation promoted by the refinement comes from an instance of a renamed *included* machine, then the first prefix to the operation name must correspond to the renaming of the instance of the machine *included* by the refinement and the name of the operation without the first prefix must correspond to an operation of the *included* machine.

Example

```

REFINEMENT
  MA_r
REFINES
  MA
INCLUDES
  b2.MC,
  MD
PROMOTES
  b2.op1,
  op3
OPERATIONS
  b2.op2 = ... ;
  op4 = ...
...
END

```

```

MACHINE
  MA
INCLUDES
  b2.MB
PROMOTES
  b2.op1,
  b2.op2
OPERATIONS
  op3 = ... ;
  op4 = ...
...
END

```

```

MACHINE
  MB
OPERATIONS
  op1 = ... ;
  op2 = ...
...
END

```

```

MACHINE
  MC
OPERATIONS
  op1 = ...
...
END

```

```

MACHINE
  MD
OPERATIONS
  op3 = ...
...
END

```

In a refinement, the formal parameters of each operation must be identical to those of the refined operation. Each formal parameter keeps the same type as defined in the abstraction.

Operation body

The body of an operation is a substitution. Only substitutions at refinement level are allowed. It is not necessary to type the formal parameters of the operation in the body of the operation. In practice, the name and the type of these parameters are determined in the machine that corresponds to the refinement and remain identical during the refinement. The operation refinement mechanism is characterized by the following properties concerning the body of the refinement operation:

1. Substitutions should be less deterministic. Therefore, where abstraction has choices, refinement should bring solutions in order to remove little by little the indeterminism. In the last refinement, implementation, indeterminism must have completely disappeared.
2. Preconditions may be weakened in refinements. In implementation, preconditions must have disappeared. In practice, preconditions are useless in refinements.
3. The structure of the substitution must therefore evolve towards the use of substitutions that are more and more concrete. Concrete substitutions are substitutions that may be executed by a computer program. In implementation, only concrete substitutions are accepted.

The properties described above allow refining an operation step by step until a computer program is obtained. It has to be proved that for each refinement, the body of the operation preserves the invariant and is consistent with what was specified in the abstraction.

Using an implementation

The OPERATIONS clause of an implementation follows the same principles as those used in a refinement, but the substitutions used in its body must obey the following principles: the substitutions used must be deterministic, the preconditions must have disappeared and the substitutions must be concrete ones, so that they can be executed by a program (refer to section 7.24, *The LOCAL_OPERATIONS Clause*).

Example

```
REFINEMENT
  MA_r
REFINES
  MA
OPERATIONS
  res_min, res_max, equal ← Compare (x1, x2)
=
  BEGIN
    IF x1 ≤ x2 THEN
      res_min, res_max := x1, x2
    ELSE
      res_min, res_max := x2, x1
    END ;
    egal := bool (x1 = x2)
  END
END
```

7.24 The LOCAL_OPERATIONS Clause

Syntax

```

Clause_local_operations ::=          "LOCAL_OPERATIONS" Operation+
Operation ::=                      Header_operation "=" Level1_substitution
Header_operation ::=                [ Ident+ "<" ] Ident+ [ "(" Ident+ ")" ]
Substitution_body_operation ::=
    Block_substitution
    | Identity_substitution
    | Becomes_equal_substitution
    | Precondition_substitution
    | Assertion_substitution
    | Substitution_limited_choice
    | If_substitution
    | Select_substitution
    | Case_substitution
    | Any_substitution
    | Let_substitution
    | Becomes_elt_substitution
    | Becomes_such_that_substitution
    | Var_substitution
    | Substitution_call

```

Restrictions

1. The formal parameters of a local operation must be distinct two by two.
2. Local operations must not be renamed.
3. Input parameters of a local operation must be typed by typing predicates in the precondition predicate located at the beginning of the body of the local operation (refer to section 3.7, *Typing operation input parameters*). These input parameters cannot be used in the precondition predicate before being typed.
4. Output parameters of a local operation must be typed in the body of the local operation (refer to section 3.9, *Typing local variables and operation output parameters*).

Description

Local operations of an implementation are local because they are usable only by the operations (local or non local) of this implementation. A local operation is specified in the LOCAL_OPERATIONS clause and implemented in the OPERATIONS clause, with the implementation of non local and non promoted operations.

The local operations share many characteristics with non local operations (refer to section 7.23, *The OPERATIONS Clause*): they can modify variables through substitutions; they can have input and output parameters. They are different than non local operations by their refinement and visibility: they are specified and implemented in the same implementation and are accessible only by the operations of the implementation in which they are defined.

It is necessary to prove that the local operation specifications preserve the invariant of *imported* machine and that the implementation of each local operation (refer to section 7.23, *The OPERATIONS Clause*) is consistent with the local operation specification.

Use

Local operations are used to reduce the size of a B project. A local operation is defined in an implementation by its specification and its implementation. As always in the B method, calls will be replaced by the specification of the local operation during the proof and by a call to their implementation in the software associated to the project.

The specification of a local operation requires abstract machine substitutions, as the specification of a non local operation. In particular, simultaneous substitutions are authorized, but not sequencing substitutions. Constants and abstract variables of the implementation refinement and of the machine instances *seen* or *imported* by the implementation are accessible in the operation specification. Moreover, *imported* variables are directly modifiable by the local operation specification.

The implementation of a local operation is located in the OPERATIONS clause, with the implementation of the machine operations which are not promoted by the implementation. They have to obey the same rules as non local operations. In particular, simultaneous substitutions is forbidden, sequencing substitutions are authorized and constants and abstract variables are not accessible in the instructions.

A local operation can be called by the implementations of the non local operation. It can be called by the implementation initialization. It can modify directly the concrete variables of the implementation and the variables of the *imported* machine instances (directly in the specification of local operations and indirectly, by operation calls in the implementation of local operations. Refer to the subsection *Equivalent model* hereafter). If a local operation is called several times, a common processing is factorized.

Example

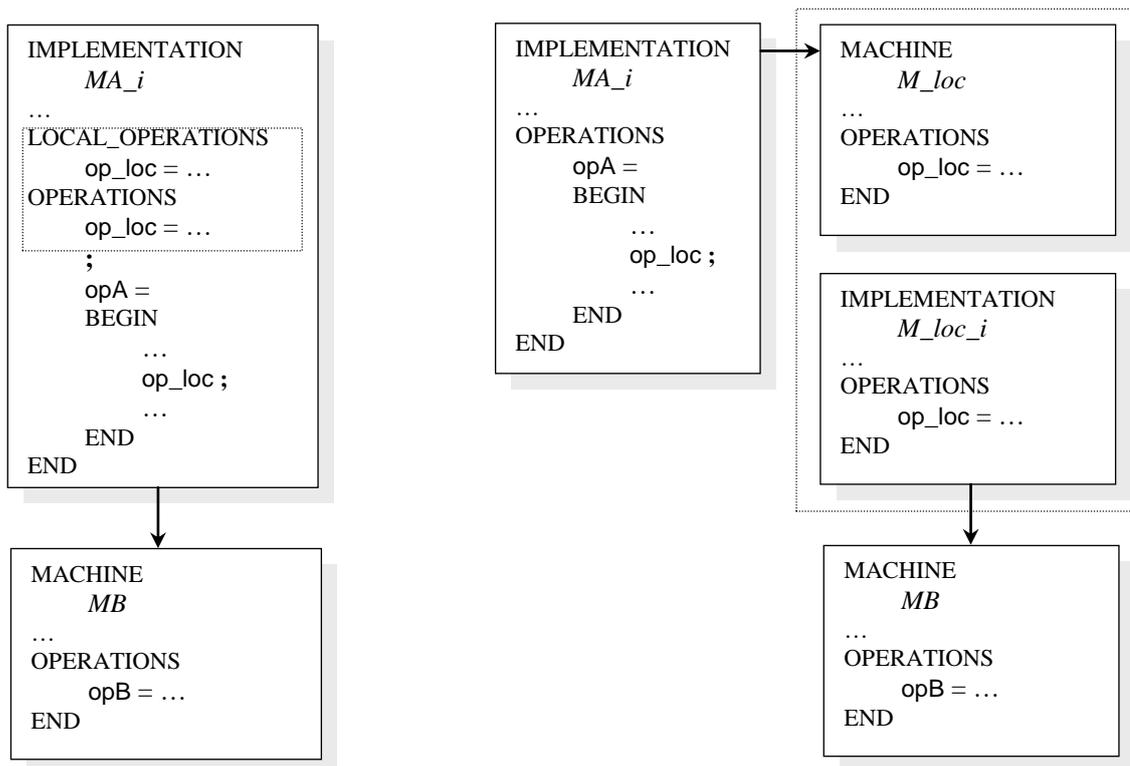
```

IMPLEMENTATION
  MA_i
  ...
LOCAL_OPERATIONS
  max_y =
  BEGIN
    x0 := max (y1, y2)
  END
OPERATIONS
  max_y =
  IF y1 ≥ y2 THEN
    x0 := y1
  ELSE
    x0 := y2
  END
  ;
  OpA =
  BEGIN
    ...
    max_y ;
    ...
  END
END

```

Equivalent model

The local operations are described by the following equivalent model. Let MA_i be an implementation that defines the local operation op_{loc} and that imports the machine MB . The general principle of the equivalent model is as follow : the implementation MA_i imports a machine M_{loc} containing the specification of op_{loc} . The machine M_{loc} is refined by the implementation M_{loc_i} which contains the implementation of op_{loc} and which *extends* MB . MA_i is obtained from MA_i by suppressing the declarations of concrete variables and the initialization. All the elements of MB are replicated in the machine M_{loc} . In the case where MB includes a machine MC , then all the data of MC are replaced in M_{loc} and one brings into effect the expansion of the calls to the MC operations. All the concrete variables in MA_i are replaced in M_{loc} . The invariant of M_{loc} is composed by the invariants of MB and MC and by the typing of concrete variables of MA_i . The initialization of M_{loc} includes the initialization of MC , then the initialization of MB , then the initialization of MA_i . At last, the instructions of the MA_i operation body which are not operation calls, are transformed into operations in M_{loc} and M_{loc_i} .



7.25 Specificities of the B0 language

B0 is the part of B language describing operations and data of implementations. B0 is the equivalent of a computer programming language, manipulating concrete data, whereas B language is a language of specification and of programming.

The concrete data present in B0 are concrete constants, concrete variables, operation input and output parameters, machine parameters, local variables, deferred sets and enumerated sets, along with their elements. This kind of data is described in section 3.4 *Types and Constraints of Concrete data*.

In order to clearly distinguish between the languages B and B0, a new vocabulary is adopted to designate productions of the B0 grammar. Concrete substitutions are called instructions (refer to section 7.25.4). Concrete predicates are called conditions (refer to section 7.25.3) and concrete expressions are called terms (refer to section 7.25.2).

7.25.1 Array controls in B0

Description

So as to guarantee that concrete arrays (refer to section 3.4 *Types and Constraints of concrete data*) can be translated, a B0 compatibility control, as defined below, is added to typing controls concerning predicates expressions and the substitutions which are to be translated .

Restrictions

1. Two concrete arrays are compatible in B0 if they have the same type and if, at syntactic level, they were typed with the same domain of definition. The domain of definition of an array is either determined directly when the array is typed in a typing predicate which explicitly defines this domain, or by inference if the array is typed with the help of another array.

Use

Two concrete arrays can be incompatible for a computing program although they are of the same type. This happens when certain index sets of the arrays are intervals of different values.

For example, the concrete arrays $Arr1 \in (1..5) \rightarrow INT$ and $Arr2 \in (1..10) \rightarrow INT$ are of the same type, but they cannot be used as a value for the same computing piece of data, because their size is different.

According to the restriction announced above, the concrete arrays $Arr1$ and $Arr2$ are not compatible in B0 since their domains of definition (1.5) and (1..10) are not syntactically distinct.

This control is a sufficient condition but not necessary so as to assure that the values of two concrete arrays are compatible. Indeed, if $c1$ and $c2$ are two concrete constants designating two equal positive integers. Then the arrays $Arr3 \in (0..c1) \rightarrow INT$ and $Arr4 \in (0..c2) \rightarrow INT$ are not compatible in B0 , even if $c1$ and $c2$ are equal.

7.25.2 TERMS

Syntax

Term ::=

```

    Simple_term
  |   Arithmetic_expression
  |   Record_term
  |   Record_term ( "" Ident ) +
Simple_term      ::=
    Ren_ident
  |   Lit_integer
  |   Lit_boolean
  |   Ren_ident ( "" Ident ) +
Lit_integer      ::=
    Literal_integer
  |   "MAXINT"
  |   "MININT"
Lit_boolean      ::=
    "FALSE"
  |   "TRUE"
Arithmetic_expression ::=
    Lit_integer
  |   Ident_ren
  |   Ren_ident (" Term "+, ")
  |   Ident_ren ( "" Ident )+
  |   Arithmetic_expression "+" Arithmetic_expression
  |   Arithmetic_expression "-" Arithmetic_expression
  |   "-" Arithmetic_expression
  |   Arithmetic_expression "x" Arithmetic_expression
  |   Arithmetic_expression "/" Arithmetic_expression
  |   Arithmetic_expression "mod" Arithmetic_expression
  |   Arithmetic_expression exp Arithmetic_expression
  |   "succ" (" Arithmetic_expression ")
  |   "pred" (" Arithmetic_expression ")
  |   (" Arithmetic_expression ")
Record_term      ::=
    "rec" (" ( [ Ident ":" ] ( Term | Array_expression ) ) +, " ")
Array_expression ::=
    Ident
  |   "{ ( Simple_term +|-> "->" Term) +, "}"
  |   Simple_term +x "x" "{ Term "}"
B0_interval      ::=
    Arithmetic_expression .. Arithmetic_expression
  |   B0_Number_set
B0_Number_set   ::=
    "NAT"
  |   "NAT1"
  |   "INT"

```

Description

Terms represent the restriction of expressions of B language that can be used in B0. Terms can be implemented by a computing program. They are used within instructions

and conditions.

Terms must be of the type of concrete variables (refer to section 3.6 Typing concrete variables)

Direct use, in the instructions, of terms requires to prove that the terms are well defined and can be correctly implemented in a classical programming language. To obtain this, the following proof obligations will have to be demonstrated :

- When data of integer type is used in a B0 expression, the proof must be made that the data belongs to INT (defined by MININT .. MAXINT) which is the set of concrete integers. Indeed, the hypothesis is made that on the target machine on which the project is being executed, it is possible to represents directly any integer between MININT and MAXINT without any risk of overflow.
- When an arithmetical operator is used in a B0 expression, it must be proved that the operands belong to the definition domain of the operator in B0 and that the result belongs to INT. The arithmetical operators which can be used in terms, along with their domains of definition are given in the array below:

B0 arithmetic expression		Condition		
B0 addition	$a + b$	$a \in \text{INT}$	$\wedge b \in \text{INT}$	$\wedge a + b \in \text{INT}$
B0 subtraction	$a - b$	$a \in \text{INT}$	$\wedge b \in \text{INT}$	$\wedge a - b \in \text{INT}$
B0 unary minus	$- a$	$a \in \text{INT}$		$\wedge - a \in \text{INT}$
B0 multiplication	$a \times b$	$a \in \text{INT}$	$\wedge b \in \text{INT}$	$\wedge a \times b \in \text{INT}$
B0 integer division	a / b	$a \in \text{INT}$	$\wedge b \in \text{INT} - \{0\}$	$\wedge a / b \in \text{INT}$
B0 modulo	$a \text{ mod } b$	$a \in \text{NAT}$	$\wedge b \in \text{NAT}_1$	$\wedge a \text{ mod } b \in \text{INT}$
B0 raise to the power	a^b	$a \in \text{INT}$	$\wedge b \in \text{NAT}$	$\wedge a^b \in \text{INT}$
B0 successor	$\text{succ}(a)$	$a \in \text{INT}$		$\wedge \text{succ}(a) \in \text{INT}$
B0 predecessor	$\text{pred}(a)$	$a \in \text{INT}$		$\wedge \text{pred}(a) \in \text{INT}$

- When an access to an element of a concrete array is made in an instruction (it is to be remembered that in B an array is a total function), it must be proved that the index used belongs to the definition domain of the array.

7.25.3 CONDITIONS

Syntax

```

Condition ::=
    Simple_term "=" Simple_term
| Simple_term "≠" Simple_term
| Simple_term "<" Simple_term
| Simple_term ">" Simple_term
| Simple_term "≤" Simple_term
| Simple_term "≥" Simple_term
| Condition "∧" Condition
| Condition "∨" Condition
| "¬" (" Condition ")
| (" Condition ")

```

Description

Conditions represent the restriction of predicates of B language that can be used in B0. Conditions can be evaluated by a computing language. They are used in B0 as branching conditions of conditional instructions IF instructions, and as exit conditions of loop instructions.

In the case of equality and inequality predicates involving arrays, it must be remembered that the arrays must, of course, be of the same type, and they must also have the same definition domain (refer to section 7.25.1 Array control in B0).

7.25.4 Instructions

Syntax

```

Instruction ::=
    Block_instruction
  | Local_variable_instruction
  | Identity_substitution
  | Becomes_equal_to_instruction
  | Operation_call_instruction
  | Conditional_instruction
  | Case_instruction
  | Assertion_instruction
  | Sequence_instruction
  | While_instruction
Block_instruction ::= "BEGIN" Instruction "END"
Local_variable_instruction ::= "VAR" Ident "+", "IN" Instruction "END"
Becomes_equal_to_instruction ::=
    Ren_ident [ "(" Term "+", ")" ] ":@" Term
  | Ren_ident ":@" Array_expression
  | Ren_ident ":@" bool "(" Condition ")"
  | Ren_ident ( "" Ident )+ ":@" Term
Operation_call_instruction ::=
    [ Ren_ident "+", " ← ] Ren_ident [ "( ( Term | Literal_string ) +", ")" ]
Sequence_instruction ::= Instruction ":", Instruction
Conditional_instruction ::=
    "IF" Condition "THEN" Instruction
  ( "ELSIF" Condition "THEN" Instruction )+
  ( "ELSE" Instruction )
  "END"
Case_instruction ::=
    "CASE" Simple_term "OF"
    "EITHER" Simple_term "+", "THEN" Instruction
  ( "OR" Simple_term "+", "THEN" Instruction )*
  [ "ELSE" Instruction ]
  "END"
  "END"
Assertion_instruction ::=
    "ASSERT" Predicate "THEN" Instruction "END"

```

```
While_instruction ::=  
    "WHILE" Condition "DO" Instruction  
    "INVARIANT" Predicate  
    "VARIANT" Expression  
    "END"
```

Description

Instructions represent a restriction of the substitutions of B language which can be implemented by a computing program. Instructions are used in the initialisation body and in the operations body. Here are the particularities of the instructions:

“becomes equal to” instruction

In an instruction “becomes equal to”, only the following affectations are allowed:

- affectation of a scalar data
- affectation of an array data, in which all the elements of this array must be given a value. The effective value can be another array data or a literal array. The arrays must of course be of the same type, but they must also have the same definition domain. (refer to 7.25.1 *Array Control in B0*).
- affectation of an array item, the indexes used to designate an array item must be terms.
- affectation of a field, or a sub-field of a record data.

Operation call instruction

In an operation call instruction, the effective input parameters can either be terms, or strings of literal characters. If an effective input or output parameter of an operation call is an array, the formal parameter and the effective parameter must of course be of the same type, but they must also have the same definition domain (refer to section 7.25.1 *Array Control in B0*).

Case instruction

In a CASE instruction, the selection expression must be a simple term.

ASSERT Instruction

In an ASSERT instruction, the introduced assertion remains a predicate because it is not used for code production, but for proof of implantation.

7.26 Identifier Anti-Collision Rules

The identifier anti-collision rules serve to avoid that in a component clause, it becomes possible to access several constituents with the same name, but which designate different constituents without knowing which one is effectively used.

The anti-collision rules are mainly dependent on the visibility rules that apply between components. In practice, if a component M_A sees an instance of machine M_B , then any data item in M_B accessible by M_A must not have the same name as any data in M_A .

The data declared in a predicate, a substitution or in an operation header are not part of anti-collision checks. In practice, they have a limited scope restricted respectively to the predicate and to the substitution and the body of the operation where they are declared. If they have the same name as an accessible constituent, then they hide it locally.

Abstract machine

With an abstract machine Mch .

List $LMch$ comprises the following identifiers of Mch :

- name of Mch ,
- name of parameters of Mch ,
- name of deferred sets and enumerated sets of Mch ,
- name of listed elements of constants of Mch ,
- name of variables of Mch ,
- name of operations of Mch .

List $LSees$ comprises the following identifiers for each *seen* machine $MSees$ by Mch :

- name of $MSees$,
- name of deferred sets and of enumerated sets of $MSees$,
- name of listed elements and of constants of $MSees$,
- name of variables of $MSees$,
- name of operations of $MSees$.

List $LInc$ comprises the following identifiers for each *included* machine $MInc$ by Mch :

- name of $MInc$,
- name of deferred sets and of enumerated sets of $MInc$,
- name of listed elements and of constants of $MInc$,
- name of variables of $MInc$,
- name of operations of $MInc$.

List $LUses$ comprises the following identifiers for each *used* machine $MUses$ by M :

- name of $MUses$,
- name of parameters of $MUses$,
- name of deferred sets and of enumerated sets of $MUses$,
- name of listed elements and of constants of $MUses$,
- name of variables of $MUses$.

Anti-Collision Rule

The names in the list $LMch \cup LSees \cup LInc \cup LUses$ must be distinct two by two.

Refinement

With a refinement *Raf* with the abstract machine *Mch*.

List $LRaf$ comprises the following identifiers of *Raf* :

- name of *Mch*,
- name of parameters of *Raf*,
- name of deferred sets and of enumerated sets of *Raf*,
- name of listed elements and of constants of *Raf*,
- name of variables of *Raf*,
- name of operations of *Raf*,
- name of abstract constants and of abstract variables in the abstraction of *Raf* that disappear in *Raf*.

List $LSees$ comprises the following identifiers for each *seen* machine $MSees$ by *Raf* :

- name of $MSees$,
- name of deferred sets and of enumerated sets of $MSees$,
- name of listed elements and of constants of $MSees$,
- name of variables of $MSees$,
- name of operations of $MSees$.

List $LInc$ comprises the following identifiers for each *included* machine $MInc$ in *Raf* :

- name of $MInc$,
- name of deferred sets and of enumerated sets of $MInc$,
- name of listed elements and of constants of $MInc$,
- name of variables of $MInc$,
- name of operations of $MInc$.

Anti-collision rules

The names in the $LRaf \cup LSees \cup LInc$ list must be distinct two by two.

Implementation

With an implementation *Imp*, the abstract machine of which is *Mch*.

List $LImp$ comprises the following identifiers of *Imp* :

- name of *Mch*,
- name of parameters of *Imp*,
- name of deferred sets, except those which are implemented by homonymy,
- name of enumerated sets and of listed elements of *Imp*, except for any enumerated set with the same name as an enumerated set in a requested machine,
- name of listed elements of *Imp*,

- name of concrete constants of *Imp*, except those valued by homonymy,
- name of abstract constants of *Imp*,
- name of concrete variables of *Imp*, except those implemented by homonymy,
- name of specific operations of *Imp*,
- name of abstract constants and of abstract variables of the abstraction of *Imp* disappear in *Imp*.

List *LImports* comprises the following identifiers for each *imported* machine *MImports* by *Imp* :

- name of *MImports*,
- name of deferred sets and listed of sets of *MImports*,
- name of listed elements and of constants of *MImports*,
- name of variables of *MImports*,
- name of operations of *MImports*.

List *LSees* comprises the following identifiers for each machine *seen* *MSees* by *Imp* :

- name of *MSees*,
- name of deferred sets and of enumerated sets of *MSees*,
- name of listed elements and of constants of *MSees*,
- name of constants of *MSees*,
- name of variables of *MSees*,
- name of operations of *MSees*.

Anti-collision rule

The names in the list $LImp \cup LImports \cup LSees$ must be distinct two by two.

8. B ARCHITECTURE

8.1 Introduction

A complete development in B corresponds to a B project. A project enables formally modeling a system of any type. The object of the B project is to produce an executable program. The functional security of this executable program is studied in detail by the B method. The construction of a B project is performed using the B module development approach.

8.2 B Module

Presentation

A B module models a sub-system; it forms a part of a B project. The modules are made up of B components. The three sorts of B components that exist are the abstract machine, refinement and implementation. A module has the following properties: it always has an abstract machine, representing the module specification. It may have an implementation and possibly some refinements. Finally, it may have related code. There are three sorts of modules defined according to their properties. These are modules developed by the successive refinements of an abstract machine, of base modules and abstract modules. These modules are described in the table below.

Properties	Module	Developed module	Base module	Abstract module
Has an abstract machine		Yes	Yes	Yes
Has an implementation and possibly refinements		Yes	No	No
Has associated code		Yes (by translation)	Yes (manually)	No

Abstract machine

An abstract machine contains the description of the specification of a B module. For this reason, B language should be regarded as a specification language. Only the abstract machine of a module is accessible by the external modules. Sometimes, the term abstract machine or more simply machine is used in place of module as a shortcut. In practice, the module name and its abstract machine together are confused with the module interface, i.e. the part that is accessible from the outside, is common to a module and its abstract machine.

An abstract machine comprises links (refer to 8.3 *Links Between Components*), a static and a dynamic part. The static part is made up of data that takes the form of sets, constants, variables or parameters and by the properties of this data. The data is a mathematical object that is part of the B language mathematical set theory (refer to chapter 5 *Expressions*), such as for example a scalar, a set, a function or a sequence. The data is encapsulated in the abstract machine. The dynamic part is used to handle the data. It is made up of the initialization that allows assigning an initial value to the variables and the operations, corresponding to services offered by the machine to handle the variables. The invariant is the property of machine variables. The invariant must be established on machine initialization and it must be preserved when a machine operation is called. Therefore, the invariant forms the statement of the machine security properties.

Refinement

The refinement of an abstract machine is a component that preserves the same interface and the same behavior as the abstract machine but which reformulates the machine data and operations using more concrete data. The refinement is also used to enrich what was specified in the abstract machine. On refinement, a machine sets and concrete data are preserved. The refinable data is refined, meaning that they may be preserved, deleted or changed in form. New data may also be introduced. The body of the operations must also be refined: each refined operation must perform what is specified in the abstraction, using refinement data and more concrete and more deterministic substitutions.

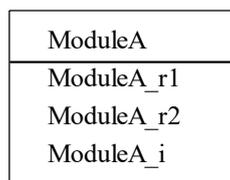
A first refinement may in turn be refined by another refinement using the principles described above. A number of refinement levels may be used in order to reformulate the abstract machine in successive steps.

Implementation

An implementation is a B component. It is the ultimate level of refinement of an abstract machine. It is written using a B language sub-set called B0 language, which has the following characteristics. The implementation data must be concrete data (scalars, arrays) that are directly implementable in a high level programming language like Ada or C++ (refer to section 7.24, *The LOCAL_OPERATIONS Clause*). The body of the operations of an implementation must be made up of concrete substitutions, called instructions, that are directly executable in a high level programming language.

Example

The diagram below graphically illustrates a developed module. ModuleA represents both the name of the module and the name of the abstract machine that represents the module specification. ModuleA_r1 and ModuleA_r2 are the names of the successive refinements of ModuleA. ModuleA_i is the name of the implementation of ModuleA.



Base module

A base module, also called a base machine, refers to a B module that has only an abstract machine. A base machine corresponds to a leaf in the import graph of a project. Unlike other modules that are refined and may be translated, this one is not translated but must have associated code that directly implements its data and its services. The base machines may be the interface with existing code or with low level functions that do not exist in B language, such as the system functions. The input/output functions are a typical example of functions interfaced by base machines.

Abstract module

An abstract module has an abstract machine that is not refined and that does not have any associated code. The only use for an abstract module in a B project is to *include* it (refer to the INCLUDES link) in an abstract machine or in a refinement without ever

importing it (refer to the IMPORTS link) into the project. It therefore forms an intermediate step for abstract reasoning.

8.3B Project

Presentation

A B project refers to a complete set of B module instances. The components of these module instances are connected by links. The links must obey certain rules.

Instantiating and renaming

A module instance is the copy of an abstract machine. Instantiating enables reusing an abstract machine a number of times in the same project. Each abstract machine instance has its own data space that contains the values of the machine modifiable data. This data is specific to the instance; the data comprises variables (refer to section 7.18, *The CONCRETE_VARIABLES Clause* and section 7.19, *The ABSTRACT_VARIABLES Clause*) and machine formal parameters (refer to section 7.5, *The CONSTRAINTS Clause*). The constants of a machine are specific to the machine as their value is identical in all machine instances. When calling up an operation from a machine instance, the values of the machine variables and parameters handled by the operation are those of the data space of the instance.

There is a distinction between abstract and concrete instances. The former are created during the specification phase and form the abstract data spaces (refer to *the INCLUDES link*). The latter are created during the implementation phase and form the concrete data spaces for the program associated to the project (refer to *the IMPORTS link*).

Each instance has its own specific name. This name may be the machine name, without renaming, an identifier, called the renaming prefix, followed by a dot and the machine name. In the case of an instance without renaming, the instance and the abstract machine have the same name, but they must not be confused. Instantiating without renaming represents the most frequent case in a B project, as the machines that are instantiated only once do not need to be renamed (it is therefore possible to choose to instance them without renaming them). On the other hand, instantiating with renaming is required as soon as a machine is instantiated a number of times, as the name of each instance of a B project machine must be unique in order to identify the data spaces.

If a component *Cmp* accesses a machine instance *InstMch*, then the name of this instance influences the name used in *Cmp* to designate the variables and operations of *InstMch*. If the instance does not use renaming, then the variables and the *InstMch* operations will be used in *Cmp* under the same name as in the abstract machine that declares it. If *InstMch* is renamed, then the name of variables and of operations used in *Cmp* must be prefixed by the renaming prefix of *InstMch* followed by a dot.

Links between components

The components of a B project may be linked together by four types of links: IMPORTS, SEES, INCLUDES and USES. All of them link a component to a machine instance they are declared in the visibility clauses for B components. Here is a brief description of them:

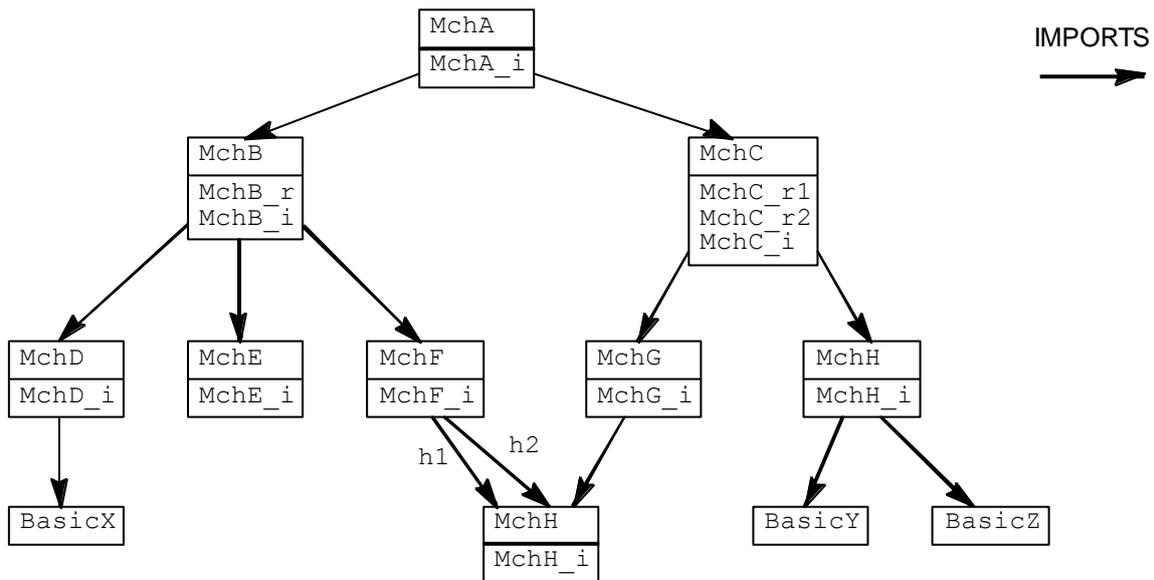
- **IMPORTS link**
The IMPORTS link between an implementation *Mimp* and a machine instance *MN* is used to create the *Mimp* instance concretely, and to gain full access to its services.

It is said that *MN* is the father of *Mimp*, as it fully controls the writing of the modifiable data in *Mimp*. *Import* is used to structure a B project into layers, since the implementation of a module is implemented by *importing* other modules that provide lower level services.

A B project *imports* graph refers to the graph made up of the B project modules and of the *import* links between the implementations of these modules. To designate which machine instance is *imported* by a link, the link should include the renaming prefix of the *imported* machine instance.

An *imports* graph must have only one root which has a specific role. This is the main machine of the project. Its operations are the project entry point. From the main machine, the *imports* graph is organized into layers that represent the levels of project breakdown into elements that are simpler and simpler. A graph leaf may be either a terminal developed module or a base machine. The import graph of a project fully describes the organization of the project executable part as each module in the graph has associated code, whether developed modules or base machines.

The diagram below shows an example of a project *imports* graph. Each module instance has 0 to *n* sons and only one father, except the main module that does not have any father.



- **SEES link**

The SEES link is a transverse reference in the B project import graph that allows a component to *see* a machine instance, i.e. to access in read mode but not in write mode, the components of the instance of the *seen* machine.

It is said that a B module is dependent on another B module, if the implementation of the first module *sees* or *imports* an instance from the second module. The dependency graph of a B project is the *imports* graph of the project to which are added the SEES links. The SEES links have the renaming suffix of the instance of the *seen* machine. This suffix may contain several successive renaming (refer to *The SEES clause* and renaming).

- **INCLUDES link**

The INCLUDES link between a component *MN* (a machine or a refinement) and an

instance of machine *Minc* enables *including* in *MN* the components of *Minc*. The *inclusion* creates the machine instance *Minc* at an abstract level.

- **USES link**

When a component *includes* a number of machine instances, the *included* machines may share the data of one of them named *Mused* via a USES link to *Mused*. The USES clause enables referencing a machine instance within several *inclusion* links.

Rules applying to links

The rules that apply to links between components in a project are given below.

Rules for the IMPORTS links

1. A machine instance must not be *imported* more than once into a project. Therefore to *import* a machine into a project a number of times, it is necessary to create a number of instances by giving them different renaming prefixes.
2. Any complete project must contain one and only one developed module that is never *imported* into the project. This is the only root of the project *imports* graph. This module is called the main project module.

Rules for the dependency links

3. Any machine instance *seen* in a project must be *imported* into the project.
4. If a machine instance is *seen* by a component, then the refinements of this component must also see this instance.
5. If a component *sees* a machine instance MA then it cannot *see* a machine instance belonging to the import sub-graph in MA. The following diagram illustrates the illegal architectures.

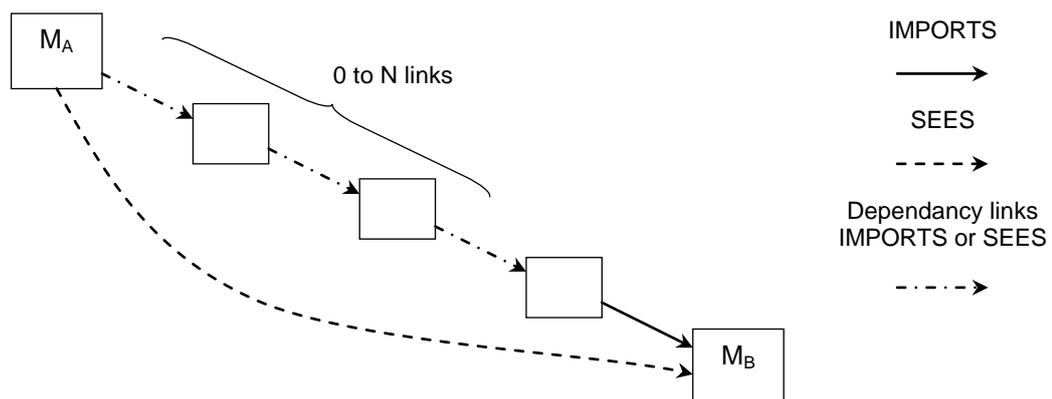


Figure 4: illegal architecture for the SEES link

6. A component must not have a several links to the same machine instance. For example, an implementation cannot *see* and *import* the same machine instance.
7. Cycles are forbidden in the project dependency graph.

8. If a component A sees a machine B, we must be able to go from A to B by following the importation tree with this rule: we must start from A, then go up at least one step et finally go down exacty one step in order to reach B. In other words, a component can only see its brother, uncle, great-uncle, great-great-uncle, etc.

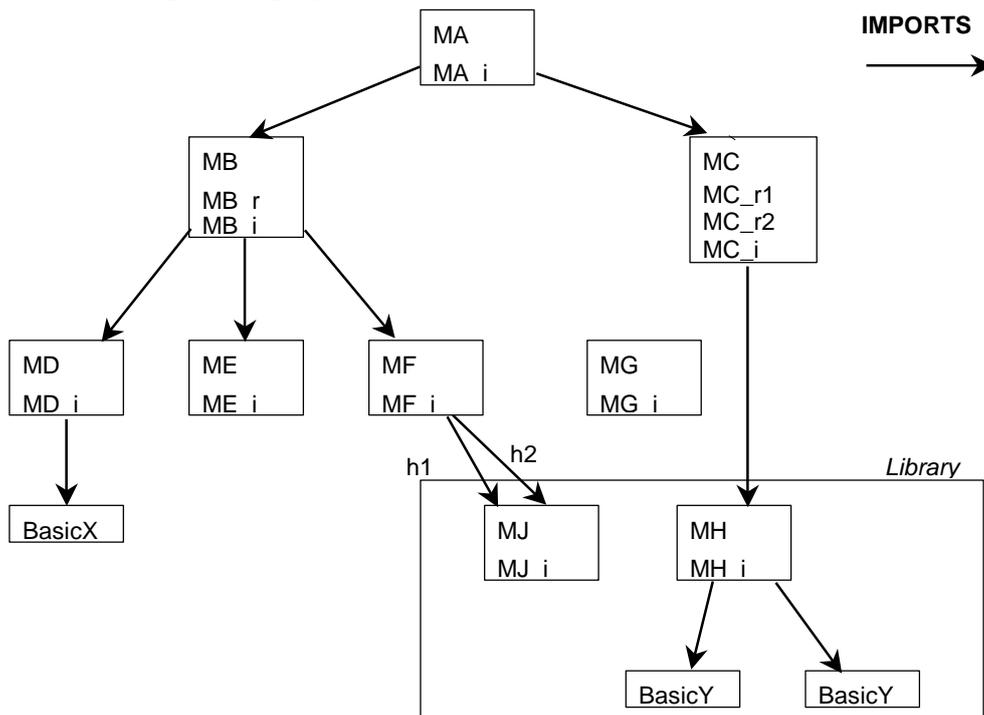
Rules for the USES links

9. If a machine *MA* uses an instance of machine *Mused*, then the project must *include* a machine that *includes* an instance of *MA* and *Mused*.

8.4 Libraries

A B library is a collection of modules that may be used in a project. The notion of individual libraries enables building whole sets of module libraries that may be reused in other projects. It also allows breaking down a project into several sub-parts, each individual sub-part being an individual library. The modules in a library may themselves in turn use other libraries.

A complete B project may become a library. However, a library does not have to correspond to a project since it may contain several main modules. The figure below shows an example of a project that uses a library.



APPENDIX

ANNEX A. RESERVED KEYWORDS AND OPERATORS

This appendix contains the description of reserved keywords and of the operators of B language, sorted by ascending ASCII order. The ASCII order is reminded below:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy{|}~
```

For each reserved keyword or operator, this chapter provides:

- its ASCII notation, which can be possibly completed by its use when there is a non-trivial correlation between the ASCII and mathematical notations (for example, in the case of the power operator, the ASCII notation $x**y$ corresponds to the mathematical notion x^y)
- its mathematical notion, if it differs from its ASCII notation.
- its priority level. The priority level corresponds to the priority level during the syntactic analysis. The higher the priority level of an operator, the more it attracts operands. For example, if the operators op_{40} and op_{250} are respectively of 40 and 250 priority, then the expression $x op_{40} y op_{250} z$ is analysed as $x op_{40} (y op_{250} z)$
- its associative properties (L for associative to the left or R for associative to the right). If two binary operators named op have the same priority, then: $x op y op z$ will be analysed as $(x op y) op z$ if op is associative to the left; and as $x op (y op z)$ if op is associative to the right.
- its description.

ASCII	Math.	Pri.	As.	Description	References
!	\forall	250		For any	4.2
"				String or definition file	5.1, 2.3
#	\exists	250		There exists	4.2
\$0				Value of data before substitution	5.1
%	λ	250		Lambda expression	5.16
&	\wedge	40	G	Conjunction (logical AND)	4.1
'		250	G	Access to a record field	5.9
(Open bracket	4.1, 5.1
)				Close bracket	4.1, 5.1
*	\times	190	G	Multiplication or Cartesian product	3.2, 5.3, 5.7
$x ** y$	x^y	200	D	Power of	5.3
+		180	G	Addition	5.3
$+-->$	\rightarrow	125	G	Partial function	5.15
$+-->>$	\twoheadrightarrow	125	G	Partial surjection	5.15
,		115	G	Comma	
-		180	G	Subtraction	5.3, 5.8
-		210		Unary minus	5.3
$-->$	\rightarrow	125	G	Total function	5.15

ASCII	Math.	Pri.	As.	Description	References
-->>	\rightarrow	125	G	Surjection	5.15
->	\rightarrow	130	G	Insert at the start of a sequence	5.15
.		220	D	Renaming or data separator used in the operators $\forall, \exists, \cup, \cap, \Sigma, \Pi, \lambda$	
..		170	G	Interval	5.7
/		190	G	Integer division	5.3
/:	\notin	160	G	Non-belonging	4.4
/ \lt :	$\not\subseteq$	110	G	Non-inclusion	4.5
/ \ll :	$\not\subset$	110	G	Strict non-inclusion	4.5
/=	\neq	160	G	Not equal	4.3
/\	\cap	160	G	Intersection	5.8
/ \	\uparrow	160	G	Restriction of a sequence to the head	5.19
:	\in	60	G	Belonging	4.4
:		120	G	Record field	5.9
::	$:\in$		G	Becomes part of (belonging)	6.12
:=			G	Becomes equal to	6.3
;		20	G	Sequencing for substitution or composition of relations	6.15, 5.11
<		160	G	Strictly less than	4.6, 2.3
<+	\triangleleft	160	G	Overload a relation	5.14
<->	\leftrightarrow	125	G	Set of relations	5.10
<-	\leftarrow	160	G	Insert at end of sequence	5.19
<--	\leftarrow		G	Operation output parameters	6.16, 7.23
<:	\subseteq	110	G	Inclusion	4.5
<<:	\subset	110	G	Strict inclusion	4.5
<<	\triangleleft	160	G	Substraction to the domain	5.14
<=	\leq	160	G	Less than or equal	4.6
<=>	\Leftrightarrow	60	G	Equivalence	4.1
<	\triangleleft	160	G	Restriction to the domain	5.14
=		60	G	Equals	4.3
==				Definition	2.3
=>	\Rightarrow	30	G	Implies	4.1
>		160	G	Strictly greater than	4.6, 2.3
>>	\rightrightarrows	125	G	Partial injection	5.15
>->	\rightarrow	125	G	Total injection	5.15
>->>	\rightarrow	125	G	Total bijection	5.15
>>	\otimes	160	G	Direct product of relations	5.11
>=	\geq	160	G	Greater than or equal	4.6
ABSTRACT_CONSTANTS				ABSTRACT_CONSTANTS clause	7.15
ABSTRACT_VARIABLES				ABSTRACT_VARIABLES clause	7.19

ASCII	Math.	Pri.	As.	Description	References
ANY				ANY substitution	6.10
ASSERT				ASSERT substitution	6.5
ASSERTIONS				ASSERTIONS clause	7.21
BE				LET substitution	6.11
BEGIN				BEGIN substitution	6.1
BOOL				Conversion of a predicate into a Boolean value	5.6
CASE				CASE substitution	6.9
CHOICE				CHOICE substitution	6.6
CONCRETE_CONSTANTS				CONCRETE_CONSTANTS clause	7.14
CONCRETE_VARIABLES				CONCRETE_VARIABLES clause	7.18
CONSTANTS				CONSTANTS clause	7.14
CONSTRAINTS				CONSTRAINTS clause	7.5
DEFINITIONS				DEFINITIONS clause	2.3
DO				WHILE substitution	6.17
EITHER				CASE substitution	6.9
ELSE				IF or CASE substitution	6.7, 6.9
ELSIF				IF substitution	6.7
END				Terminator of clauses or of substitutions BEGIN, PRE, ASSERT, CHOICE, IF, SELECT, ANY, LET, VAR, CASE and WHILE	
EXTENDS				clause EXTENDS	7.11
FALSE				Literal Boolean constant "false"	5.2
FIN	F			Set of finite sub-sets	5.7
FIN1	F_1			Set of finite non empty sub-sets	5.7
IF				Substitution IF	6.7
IMPLEMENTATION				IMPLEMENTATION clause	7.4
IMPORTS				IMPORTS clause	7.7
IN				BE or VAR substitution	6.11, 6.17
INCLUDES				INCLUDES clause	7.9
INITIALISATION				INITIALISATION clause	7.22
INT				Set of implementable relative integers	5.6
INTEGER	\mathbb{Z}			Set of relative integers	5.6
INTER	\cap			Quantified intersection	5.8
INVARIANT				INVARIANT clause or WHILE substitution	7.20, 6.17
LET				LET substitution	6.11, 6.14
LOCAL_OPERATIONS				LOCAL_OPERATIONS clause	7.24
MACHINE				MACHINE clause	7.1
MAXINT				Largest implementable integer	5.3
MININT				Smallest implementable integer	5.3

ASCII	Math.	Pri.	As.	Description	References
NAT				Set of implementable natural integers	5.6
NAT1	NAT_1			Set of non-empty implementable natural integers	5.6
NATURAL	\mathbb{N}			Set of natural integers	5.6
NATURAL1	\mathbb{N}_1			Set of non-empty natural integers	5.6
OF				CASE substitution	6.9
OPERATIONS				OPERATIONS clause	7.23
OR				CHOICE or CASE substitution	6.6, 6.9
PI	Π			Quantified integer product	5.4
POW	\mathbb{P}			Set of sub-sets	5.7
POW1	\mathbb{P}_1			Set of non-empty sub-sets	5.7
PRE				Precondition substitution	6.4
PROMOTES				PROMOTES clause	7.10
PROPERTIES				PROPERTIES clause	7.16
REFINES				REFINES clause	7.6
REFINEMENT				REFINEMENT clause	7.3
SEES				SEES clause	7.8
SELECT				SELECT clause	6.8
SETS				SETS clause	7.13
SIGMA	Σ			Quantified product	5.4
STRING				Set of character strings	5.6
THEN				Precondition substitution, ASSERT, IF or CASE	7.10, 6.5, 6.7, 6.9
TRUE				Literal Boolean constant "true"	5.2
UNION	\cup			Quantified union	5.8
USES				USES clause	7.12
VALUES				VALUES clause	7.17
VAR				VAR substitution	6.14
VARIANT				WHILE substitution	6.17
VARIABLES				VARIABLES clause	7.19
WHEN				SELECT substitution	6.8
WHERE				ANY substitution	6.10
WHILE				WHILE substitution	6.17
[Start of sequence	5.13, 5.17
[]				Empty sequence	5.17
\ /	\cup	160	G	Union	5.8
\ /	\downarrow	160	G	Restrict a sequence to the end	5.19
]				End of sequence	5.13, 5.17
^	\wedge	160	G	Concatenate sequences	5.19
arity				Tree node arity	5.22
bin				Binary tree in extension	5.20

ASCII	Math.	Pri.	As.	Description	References
bool				Predicate boolean cast	5.2
btree				Binary trees	5.20
card				Cardinal	5.4
closure(R)	R^*			Reflexive closure of a relation	5.12
closure1(R)	R^+			Closure of a relation	5.12
conc				Concatenation of a succession	5.19
const				Tree constructor	5.21
dom				Domain of a function	5.13
father				Father of a tree node	5.22
first				First element in a sequence	5.18
fnc				Transformed into a function	5.16
front				Front of a sequence	5.18
id				Function identity	5.11
infix				Infix formulae of a tree	5.20
inter				General intersection	5.8
iseq				Set of injective sequences	5.17
iseq1	iseq ₁			Set of injective non-empty sequences	5.17
iterate(R, n)	R^n			Iteration of a relation	5.12
last				Last element in a sequence	5.18
left				Left tree	5.20
max				Maximum in a set of integers	5.4
min				Minimum in a set of integers	5.4
mirror				Mirror of a tree	5.21
mod		190	G	Modulo	5.3
not	\neg			Logical no	4.1
or	\vee	40	G	Disjunction (logical OR)	4.1
perm				Set of permutations (bijective sequences)	5.17
postfix				Postfix formulae of a tree	5.21
pred				Predecessor of an integer	5.3
prefix				Prefix formulae of a tree	5.21
prj1	prj ₁			First projection of a relation	5.11
prj2	prj ₂			Second projection of a relation	5.11
ran				Range of a relation	5.13
rank				Rank of a tree node	5.22
rec				Record in extension	5.9
rel				Set of relations	5.16
rev				Reverse of a sequence	5.18
right				Right tree	5.20
seq				Set of sequences	5.17
seq1				Set of non-empty sequences	5.17

ASCII	Math.	Pri.	As.	Description	References
size				Size of a sequence	5.18
sizet				Size of a tree	5.21
skip				Null substitution	6.2
son				i^{th} son of a tree	5.22
sons				Sons of a tree node	5.21
struct				Set of records	5.9
subtree				Subtree of a tree	5.22
succ				Successor	5.3
tail				Tail of a sequence	5.18
top				Top of a tree	5.21
tree				Trees	5.20
union				Generalized union	5.8
{				Start of set	5.7
{ }	\emptyset			Empty set	5.6
		10	G	Vertical bar used in $\forall, \exists, \cup, \cap, \Sigma, \Pi, \lambda$	
->	\mapsto	160	G	Maplet	5.5
>	\triangleright	160	G	Restriction to the range	5.14
>>	$\triangleright\triangleright$	160	G	Substraction to the range	5.14
		20	G	Simultaneous substitutions parallel product of relations	5.11
}				End of set	5.7
$r\sim$	r^{-1}	230	G	Reverse relation	5.11

ANNEX B. GRAMMAR

This Appendix covers B language grammar, the grammar of typing predicates and of types. The lexical and syntax conventions used to describe this grammar is defined in chapter 2.

B.1 B Language Grammar

B.1.1 Initial Axiom

Component ::=

```

Machine_abstract
| Refinement
| Implementation

```

B.1.2 Clauses

Machine_abstract ::=

```

"MACHINE" Header
Clause_machine_abstract*
"END"

```

Clause_machine_abstract ::=

```

Clause_constraints
| Clause_sees
| Clause_includes
| Clause_promotes
| Clause_extends
| Clause_uses
| Clause_sets
| Clause_concrete_constants
| Clause_abstract_constants
| Clause_properties
| Clause_concrete_variables
| Clause_abstract_variables
| Clause_invariant
| Clause_assertions
| Clause_initialization
| Clause_operations

```

Header ::=

```

Ident [ "(" Ident+ "," "]" ]

```

Refinement ::=

```

"REFINEMENT" En-tête
Clause_refines
Clause_refinement*
"END"

```

Clause_refinement ::=

```

Clause_sees
| Clause_includes
| Clause_promotes
| Clause_extends
| Clause_sets
| Clause_concrete_constants

```

```

|   Clause_abstract_constants
|   Clause_properties
|   Clause_concrete_variables
|   Clause_abstract_variables
|   Clause_invariant
|   Clause_assertions
|   Clause_initialization
|   Clause_operations

```

Implementation ::=

```

"IMPLEMENTATION" Header
Clause_refines
Clause_implementation*
"END"

```

Clause_implementation ::=

```

Clause_sees
|   Clause_imports
|   Clause_promotes
|   Clause_extends_B0
|   Clause_sets
|   Clause_concrete_constants
|   Clause_properties
|   Clause_values
|   Clause_concrete_variables
|   Clause_invariant
|   Clause_assertions
|   Clause_initialization_B0
|   Clause_operations_B0

```

Clause_constraints ::=

```

"CONSTRAINTS" Predicate+^"

```

Clause_refines ::=

```

"REFINES" Ident

```

Clause_IMPORTS ::=

```

IMPORTS ( [Ident "." ] Ident [ "(" Instanciacion_B0+ " " ] )+ ;

```

Instanciacion_B0 ::=

```

Term
|   Number_set_B0
|   BOOL
|   Interval

```

Clause_sees ::=

```

"SEES" (Ident+ "." )+ ;

```

Clause_includes ::=

```

"INCLUDES" ( [Ident "." ] Ident [ "(" Instanciacion+ " " ] )+ ;

```

Instanciacion ::=

```

Terme
|   Number_set
|   "BOOL"
|   Interval

```

Clause_promotes ::=

```

"PROMOTES" ( Ident+ "." )+ ;

```

Clause_EXTENDS ::=

```

"EXTENDS" ( [Ident "." ] Ident [ "(" Instantiating+ " " ] )+ ;

```

Clause_EXTENDS_B0 ::=

```

"EXTENDS" ( [Ident "." ] Ident [ "(" Instanciing_B0+ " " ] )+ ;

```

```

Clause_uses ::=
    "USES" ( [Ident"."]Ident )+
Clause_sets ::=
    "SETS" Set+
Set ::=
    Ident
    | Ident "=" "{" Ident+ ";" }"
Clause_concrete_constants ::=
    "CONCRETE_CONSTANTS" Ident+
    | "CONSTANTS" Ident+
Clause_abstract_constants ::=
    "ABSTRACT_CONSTANTS" Ident+
Clause_properties ::=
    "PROPERTIES" Predicate+^
Clause_values ::=
    "VALUES" Valuing+
Valuation ::=
    Ident "=" Term
    | Ident "=" "Bool" "(" Condition ")"
    | Ident "=" Expr_array
    | Ident "=" Interval_B0
Clause_concrete_variables ::=
    "CONCRETE_VARIABLES" ( Ident+ )+
Clause_abstract_variables ::=
    "ABSTRACT_VARIABLES" Ident+
    | "VARIABLES" Ident+
Clause_invariant ::=
    "INVARIANT" Predicate+^
Clause_assertions ::=
    "ASSERTIONS" Predicate+
Clause_initialization ::=
    "INITIALISATION" Substitution
Clause_initialization_B0 ::=
    "INITIALISATION" Instruction
Clause_operations ::=
    "OPERATIONS" Operation+
Operation ::=
    Header_operation "=" Level1_substitution
Header_operation ::=
    [ Ident+ ";" "<" ] Ident+ [ "(" Ident+ ";" ")" ]
Clause_operations_B0 ::=
    "OPERATIONS" Operation_B0+
Operation_B0 ::=
    Header_operation "=" Level1_instruction

```

B.1.3 Terms and Groups of Expressions

```

Term ::=
  Simple_term
  | Ident+ (" Term+; ")
  | Arithmetical_expression

Simple_term ::=
  Ident+
  | Integer_lit
  | Boolean_lit

Integer_lit ::=
  Integer_literal
  | "MAXINT"
  | "MININT"

Boolean_lit ::= "FALSE"
  | "TRUE"

Arithmetical_expression ::=
  Integer_lit
  | Ident+
  | Arithmetical_expression "+" Arithmetical_expression
  | Arithmetical_expression "-" Arithmetical_expression
  | "-" Arithmetical_expression
  | Arithmetical_expression "×" Arithmetical_expression
  | Arithmetical_expression "/" Arithmetical_expression
  | Arithmetical_expression "mod" Arithmetical_expression
  | Arithmetical_expression Expression_arithmétique
  | "succ" "(" Arithmetical_expression ")"
  | "pred" "(" Arithmetical_expression ")"
  | "floor" "(" Expression_arithmétique ")"
  | "ceiling" "(" Expression_arithmétique ")"
  | "real" "(" Expression_arithmétique ")"

Expr_array ::=
  Ident
  | "{" ( Simple_term+ "↦" Term )+ "}"
  | ( Range+ "×" "{" Term } )+ "v"

Range ::=
  Ident
  | Interval_B0
  | "{" Simple_term+ "}"

Interval_B0 ::=
  Arithmetical_expression ".." Arithmetical_expression
  | Number_set_B0

Number_set_B0 ::=
  "NAT"
  | "NAT1"
  | "INT"

```

B.1.4 Conditions

```

Condition ::=
    Simple_term "=" Simple_term
  |
    Simple_term "≠" Simple_term
  |
    Simple_term "<" Simple_term
  |
    Simple_term ">" Simple_term
  |
    Simple_term "≤" Simple_term
  |
    Simple_term "≥" Simple_term
  |
    Condition "^" Condition
  |
    Condition "∨" Condition
  |
    "-" "(" Condition ")"

```

B.1.5 Instructions

```

Instruction ::=
    Level1_instruction
  |
    Sequence_instruction
Level1_instruction ::=
    Block_instruction
  |
    Var_instruction
  |
    Identity_substitution
  |
    Becomes_equal_instruction
  |
    Callup_instruction
  |
    If_instruction
  |
    Case_instruction
  |
    Assert_instruction
  |
    While_substitution
Block_instruction ::=
    "BEGIN" Instruction "END"
Var_instruction ::=
    "VAR" Ident+ ":" "IN" Instruction "END"
Becomes_equal_instruction ::=
    Ident+ [ "(" Term+ ")" ] "!=" Term
  |
    Ident+ "!=" Expr_array
  |
    Ident+ "!=" "bool" "(" Condition ")"
Callup_instruction ::=
    [ (Ident+)+ "←" ] Ident+ [ "(" (Term | String_lit)+ ")" ]
Sequence_instruction ::=
    Instruction ";" Instruction
If_instruction ::=
    "IF" Condition "THEN" Instruction
    ( "ELSIF" Condition "THEN" Instruction )*
    [ "ELSE" Instruction ]
    "END"
Case_instruction ::=
    "CASE" Simple_term "OF"
    "EITHER" Simple_term+ "THEN" Instruction
    ( "OR" Simple_term+ "THEN" Instruction )*
    [ "ELSE" Instruction ]
    "END"
    "END"

```

B.1.6 Predicates

```

Predicate ::=
    Bracketed_predicate
    | Conjunction_predicate
    | Negation_predicate
    | Disjunction_predicate
    | Implication_predicate
    | Equivalence_predicate
    | Universal_predicate
    | Existential_predicate
    | Equals_predicate
    | Unequals_predicate
    | Belongs_predicate
    | Non_belongs_predicate
    | Inclusion_predicate
    | Strict_inclusion_predicate
    | Non_inclusion_predicate
    | Non_strict_inclusion_predicate
    | Less_than_or_equal_predicate
    | Strictly_less_than_predicate
    | Greater_or_equal_predicate
    | Strictly_greater_predicate

Bracketed_predicate ::=
    "(" Predicate ")"

Conjunction_predicate ::=
    Predicate "∧" Predicate

Negation_predicate ::=
    "¬" "(" Predicate ")"

Disjunction_predicate ::=
    Predicate "∨" Predicate

Implication_predicate ::=
    Predicate "⇒" Predicate

Equivalence_predicate ::=
    Predicate "⇔" Predicate

Predicate_universal ::=
    "∀" List_ident "." "(" Predicate "⇒" Predicate ")"

Existential_predicate ::=
    "∃" List_ident "." "(" Predicate ")"

Equals_predicate ::=
    Expression "=" Expression

Predicate_unequal ::=
    Expression "≠" Expression

Belongs_predicate ::=
    Expression "∈" Expression

Non_belongs_predicate ::=
    Expression "∉" Expression

Predicate_includes ::=
    Expression "⊆" Expression

Predicate_includes_strictly ::=
    Expression "⊂" Expression

Non_inclusion_predicate ::=
    Expression "⊄" Expression

Non_inclusion_predicate_strict ::=
    Expression "⊈" Expression

```

Less_than_or_equal_predicate ::=
 Expression "≤" Expression
Strictly_less_than_predicate ::=
 Expression "<" Expression
Preedicate_greater_than_or_equal ::=
 Expression "≥" Expression
Strictly_greater_predicate_than ::=
 Expression ">" Expression

B.1.7 Expressions

Expression ::=
 Expressions_primary
 | *Expressions_Boolean*
 | *Expressions_arithmetical*
 | *Expressions_of_couples*
 | *Expressions_of_sets*
 | *Construction_of_sets*
 | *Expressions_of_relations*
 | *Expressions_of_functionss*
 | *Construction_of_functionss*
 | *Expressions_of_sequences*
Expression_primary ::=
 Data
 | *Expr_bracketed*
Expressions_boolean ::=
 Boolean_lit
 | *Conversion_Bool*
Expressions_arithmetical ::=
 Integer_lit
 | *Addition*
 | *Difference*
 | *Unary_minus*
 | *Product*
 | *Division*
 | *Modulo*
 | *Power_of*
 | *Successor*
 | *Predecessor*
 | *Maximum*
 | *Minimum*
 | *Cardinal*
 | *Generalized_sum*
 | *Generalized_product*
 | *Floor*
 | *Ceiling*
 | *Real_conversion*
Expressions_of_couple ::=
 Couple
Expressions_of_sets ::=
 Empty_set
 | *Number_set*
 | *Boolean_set*
 | *Strings_set*

```

Construction_of_sets ::=
|   Product
|   Comprehension_set
|   Subsets
|   Finite_subsets
|   Set_extension
|   Interval
|   Difference
|   Union
|   Intersection
|   Generalized_union
|   Generalized_intersection
|   Quantified_union
|   Quantified_intersection

Expressions_of_relations ::=
|   Relations
|   Identity
|   Reverse
|   First_projection
|   Second_projection
|   Composition
|   Direct_product
|   Parallel_product
|   Iteration
|   Reflexive_closure
|   Closure
|   Domain
|   Range
|   Image
|   Restriction
|   Antirestriction
|   Corestriction
|   Anticorestriction
|   Overwrite

Expressions_of_functionss ::=
|   Partial_functions
|   Total_functions
|   Partial_injections
|   Total_injections
|   Partial_surjections
|   Total_surjections
|   Total_bijections

Construction_of_functionss ::=
|   Lambda_expression
|   Function_constant
|   Evaluation_functions
|   Transformed_function
|   Transformed_relation

Expressions_of_sequences ::=
|   Sequences
|   Non_empty_sequences
|   Injective_sequences
|   Non_empty_inj_sequences
|   Permutations

```

<i>Construction_of_sequences</i>	::=	<ul style="list-style-type: none"> <i>Empty_sequence</i> <i>Sequence_extension</i> <i>Sequence_size</i> <i>Sequence_first_element</i> <i>Sequence_last_element</i> <i>Head_sequence</i> <i>Queue_sequence</i> <i>Reverse_sequence</i> <i>Concatenation</i> <i>Insert_start</i> <i>Insert_tail</i> <i>Restrict_head</i> <i>Restrict_tail</i> <i>Generalized_concat</i>
<i>Data</i>	::=	<ul style="list-style-type: none"> Ident+." Ident+".\$0
<i>Expr_bracketed</i>	::=	"(" <i>Expression</i> ")"
<i>Boolean_lit</i>	::=	<ul style="list-style-type: none"> FALSE TRUE
<i>Conversion_Bool</i>	::=	bool "(" <i>Predicate</i> ")"
<i>Integer_lit</i>	::=	<ul style="list-style-type: none"> Integer_literal MAXINT MININT
<i>Addition</i>	::=	<i>Expression</i> + <i>Expression</i>
<i>Difference</i>	::=	<i>Expression</i> - <i>Expression</i>
<i>Unary_minus</i>	::=	- <i>Expression</i>
<i>Product</i>	::=	<i>Expression</i> × <i>Expression</i>
<i>Division</i>	::=	<i>Expression</i> / <i>Expression</i>
<i>Modulo</i>	::=	<i>Expression</i> mod <i>Expression</i>
<i>Power_of</i>	::=	<i>Expression</i> <i>Expression</i>
<i>Successor</i>	::=	succ ["(" <i>Expression</i> ")"]
<i>Predecessor</i>	::=	pred ["(" <i>Expression</i> ")"]
<i>Maximum</i>	::=	max "(" <i>Expression</i> ")"
<i>Minimum</i>	::=	min "(" <i>Expression</i> ")"
<i>Cardinal</i>	::=	card "(" <i>Expression</i> ")"
<i>Floor</i>	::=	floor "(" <i>Expression</i> ")"
<i>Ceiling</i>	::=	ceiling "(" <i>Expression</i> ")"
<i>Real_conversion</i>	::=	real "(" <i>Expression</i> ")"
<i>Generalized_sum</i>	::=	Σ List_ident. "(" <i>Predicate</i> + \wedge <i>Expression</i> ")"
<i>Generalized_product</i>	::=	Π List_ident . "(" <i>Predicate</i> + \wedge <i>Expression</i> ")"
<i>Couple</i>	::=	<ul style="list-style-type: none"> <i>Expression</i> \mapsto <i>Expression</i> <i>Expression</i> , <i>Expression</i>
<i>Empty_set</i>	::=	\emptyset
<i>Number_set</i>	::=	<ul style="list-style-type: none"> \mathbb{Z} \mathbb{N} \mathbb{N}_1

		NAT
		NAT ₁
		INT
<i>Boolean_set</i>	::=	BOOL
<i>Strings_set</i>	::=	STRING
<i>Product</i>	::=	<i>Expression</i> × <i>Expression</i>
<i>Comprehension_set</i>	::=	{ <i>Ident</i> ⁺ , " <i>Predicate</i> ^{+^} }
<i>Subsets</i>	::=	\mathbb{P} (" <i>Expression</i> ")
		\mathbb{P}_1 (" <i>Expression</i> ")
<i>Finite_subsets</i>	::=	\mathbb{F} (" <i>Expression</i> ")
		\mathbb{F}_1 (" <i>Expression</i> ")
<i>Set_extension</i>	::=	{ <i>Expression</i> ⁺ , " }
<i>Interval</i>	::=	<i>Expression</i> .. <i>Expression</i>
<i>Difference</i>	::=	<i>Expression</i> - <i>Expression</i>
<i>Union</i>	::=	<i>Expression</i> ∪ <i>Expression</i>
<i>Intersection</i>	::=	<i>Expression</i> ∩ <i>Expression</i>
<i>Generalized_union</i>	::=	union (" <i>Expression</i> ")
<i>Generalized_intersection</i>	::=	inter (" <i>Expression</i> ")
<i>Quantified_union</i>	::=	\bigcup <i>List_ident.</i> (" <i>Predicate</i> ^{+^} <i>Expression</i> ")
<i>Quantified_intersection</i>	::=	\bigcap <i>List_ident.</i> (" <i>Predicate</i> ^{+^} <i>Expression</i> ")
<i>Relations</i>	::=	<i>Expression</i> ↔ <i>Expression</i>
<i>Identity</i>	::=	id (" <i>Expression</i> ")
<i>Reverse</i>	::=	<i>Expression</i> -1
<i>First_projection</i>	::=	prj1 (" <i>Expression</i> , <i>Expression</i> ")
<i>Second_projection</i>	::=	prj2 (" <i>Expression</i> , <i>Expression</i> ")
<i>Composition</i>	::=	<i>Expression</i> ; <i>Expression</i>
<i>Direct_product</i>	::=	<i>Expression</i> ⊗ <i>Expression</i>
<i>Parallel_product</i>	::=	<i>Expression</i> <i>Expression</i>
<i>Iteration</i>	::=	<i>Expression</i> <i>Expression</i>
<i>Reflexive_closure</i>	::=	<i>Expression</i> [*]
<i>Closure</i>	::=	<i>Expression</i> ⁺
<i>Domain</i>	::=	dom (" <i>Expression</i> ")
<i>Range</i>	::=	ran (" <i>Expression</i> ")
<i>Image</i>	::=	<i>Expression</i> [<i>Expression</i>]
<i>Domain_restriction</i>	::=	<i>Expression</i> ◁ <i>Expression</i>
<i>Domain_subtraction</i>	::=	<i>Expression</i> ◁ <i>Expression</i>
<i>Range_restriction</i>	::=	<i>Expression</i> ▷ <i>Expression</i>
<i>Range_subtraction</i>	::=	<i>Expression</i> ▷ <i>Expression</i>
<i>Overwrite</i>	::=	<i>Expression</i> ◀ <i>Expression</i>
<i>Partial_functions</i>	::=	<i>Expression</i> → <i>Expression</i>
<i>Total_functions</i>	::=	<i>Expression</i> → <i>Expression</i>
<i>Partial_injections</i>	::=	<i>Expression</i> ↗ <i>Expression</i>

<i>Total_injections</i>	::=	<i>Expression</i> \rightarrow <i>Expression</i>
<i>Partial_surjections</i>	::=	<i>Expression</i> \rightarrow <i>Expression</i>
<i>Total_surjections</i>	::=	<i>Expression</i> \rightarrow <i>Expression</i>
<i>Total_bijections</i>	::=	<i>Expression</i> \rightarrow <i>Expression</i>
<i>Lambda_expression</i>	::=	λ <i>List_ident</i> . "(" <i>Predicate</i> <i>Expression</i> ")"
<i>Evaluation_functions</i>	::=	<i>Expression</i> "(" <i>Expression</i> ")"
<i>Transformed_function</i>	::=	fnc "(" <i>Expression</i> ")"
<i>Transformed_relation</i>	::=	rel "(" <i>Expression</i> ")"
<i>Sequences</i>	::=	seq "(" <i>Expression</i> ")"
<i>Non_empty_sequences</i>	::=	seq1 "(" <i>Expression</i> ")"
<i>Injective_sequences</i>	::=	iseq "(" <i>Expression</i> ")"
<i>Non_empty_inj_sequences</i>	::=	iseq1 "(" <i>Expression</i> ")"
<i>Permutations</i>	::=	perm "(" <i>Expression</i> ")"
<i>Empty_sequence</i>	::=	[]
<i>Sequence_extension</i>	::=	[<i>Expression</i> +, "]"
<i>Sequence_size</i>	::=	size "(" <i>Expression</i> ")"
<i>Sequence_first_element</i>	::=	first "(" <i>Expression</i> ")"
<i>Sequence_last_element</i>	::=	last "(" <i>Expression</i> ")"
<i>Sequence_front</i>	::=	front "(" <i>Expression</i> ")"
<i>Sequence_tail</i>	::=	tail "(" <i>Expression</i> ")"
<i>Reverse_sequence</i>	::=	rev "(" <i>Expression</i> ")"
<i>Concatenation</i>	::=	<i>Expression</i> $\hat{\ } $ <i>Expression</i>
<i>Insert_front</i>	::=	<i>Expression</i> \rightarrow <i>Expression</i>
<i>Insert_tail</i>	::=	<i>Expression</i> \leftarrow <i>Expression</i>
<i>Restrict_front</i>	::=	<i>Expression</i> \uparrow <i>Expression</i>
<i>Restrict_tail</i>	::=	<i>Expression</i> \downarrow <i>Expression</i>
<i>Generalized_concat</i>	::=	conc "(" <i>Expression</i> ")"

B.1.8 Substitutions

```

Substitution ::=
    Level1_substitution
    | Sequence_substitution
    | Simultaneous_substitution
Level1_substitution ::=
    Block_substitution
    | Identity_substitution
    | Becomes_equal_substitution
    | Precondition_substitution
    | Assertion_substitution
    | Choice_limited_substitution
    | If_substitution
    | Select_substitution
    | Case_substitution
    | Any_substitution
    | Let_substitution
    | Becomes_elt_substitution
    | Becomes_such_that_substitution
    | Var_substitution
    | Call_up_substitution
    | While_substitution
Block_substitution ::=
    "BEGIN" Substitution "END"
Identity_substitution ::=
    "skip"
Substitution_become_equal ::=
    ( Ident+ )+ "==" Expression+
    | Ident+ "(" Expression+ ")" "==" Expression
Precondition_substitution ::=
    "PRE" Predicate "THEN" Substitution "END"
Assertion_substitution ::=
    "ASSERT" Predicate "THEN" Substitution "END"
Substitution_limited_choice ::=
    "CHOICE" Substitution ( "OR" Substitution )* "END"
If_substitution ::=
    "IF" Predicate "THEN" Substitution
    [ "ELSIF" Predicate "THEN" Substitution ]*
    [ "ELSE" Substitution ]
    "END"
Select_substitution ::=
    "SELECT" Predicate "THEN" Substitution
    ( "WHEN" Predicate "THEN" Substitution )*
    [ "ELSE" Substitution ]
    "END"
Substitute_case ::=
    "CASE" Expression "OF"
    "EITHER" Simple_term+ "THEN" Substitution
    ( "OR" Simple_term+ "THEN" Substitution )+
    [ "ELSE" Substitution ]
    "END"
    "END"
Any_substitution ::=
    "ANY" Ident+ "WHERE" Predicate "THEN" Substitution "END"

```

```

Let_substitution ::=
    "LET" Ident+ "BE"
    ( Ident "=" Expression )+ ^
    "IN" Substitution "END"

Becomes_elt_substitution ::=
    ( Ident+ ":" )+ ":" ∈ Expression

Becomes_such_that_substitution ::=
    ( Ident+ ":" )+ ":" "(" Predicate ")"

Var_substitution ::=
    "VAR" Ident+ "IN" Substitution "END"

Sequence_substitution ::=
    Substitution ";" Substitution

Substitution_callup ::=
    [ ( Ident+ ":" )+ ":" "←" ] Ident+ [ "(" Expression+ ")" ]

While_substitution ::=
    "WHILE" Condition "DO" Instruction
    "INVARIANT" Predicate
    "VARIANT" Expression
    "END"

Simultaneous_substitution ::=
    Substitution "||" Substitution

```

B.1.9 Useful Syntax Rule

```

List_ident ::=
    Ident
    | ( Ident+ "," )

```

B.1.10 Grammar of Typing Predicates

```

Typing_abstract_data ::=
    Ident+ ":" ∈ Expression+ x+
    | Ident "⊆" Expression
    | Ident "⊂" Expression
    | Ident "=" Expression

Typing_concrete_cts ::=
    Ident+ ":" ∈ Typing_belonging_concrete_cts+ x+
    | Ident "=" Typing_equals_concrete_cts
    | Ident "⊆" Simple_set
    | Ident "⊂" Simple_set

Typing_belonging_concrete_data ::=
    Simple_set
    | ( Simple_set )+ x+ "→" Simple_set
    | ( Simple_set )+ x+ "→>" Simple_set
    | ( Simple_set )+ x+ "⇒" Simple_set
    | ( Simple_set )+ x+ "⇒>" Simple_set
    | "{" Simple_term+ ":" "}"

Predicate_typing_equals_concrete_cts ::=
    | Term
    | Expr_array
    | Interval
    | Number_set_B0

```

```

Simple_set ::=
    Number_set_B0
    | "BOOL"
    | Interval
    | Ident
Number_set_B0 ::=
    "NAT"
    | "NAT1"
    | "INT"
Expr_array ::=
    Ident
    | "{" ( Simple_term+ "↦" Term )+ "}"
    | ( Range+ "x" "{" Term } )+ "∪"
Range ::=
    Ident
    | Interval
    | "{" Simple_term+ "}"
Interval ::=
    Expression ".." Expression
Typing_concrete_var ::=
    Ident+ "∈" Typing_belonging_concrete_var+
    | Ident "=" Term
Typing_belonging_concrete_data ::=
    Simple_set
    | (Simple_set)+ "→" Simple_set
    | (Simple_set)+ ">→" Simple_set
    | (Simple_set)+ "→" Simple_set
    | (Simple_set)+ ">⇒" Simple_set
    | "{" Simple_term+ "}"
Typing_param_input ::=
    Ident+ "∈" Typing_belonging_input_param+
    | Ident "=" Term
Typing_belonging_input_param ::=
    Simple_set
    | (Simple_set)+ "→" Simple_set
    | (Simple_set)+ ">→" Simple_set
    | (Simple_set)+ "⇒" Simple_set
    | (Simple_set)+ ">⇒" Simple_set
    | "{" Simple_term+ "}"
    | Set_string
Set_string ::=
    "STRING"
Typing_param_mch ::=
    Ident+ "∈" Typing_belonging_param_mch+
    | Ident+ "=" Term+
Typing_belonging_param_mch ::=
    Number_set
    | "BOOL"
    | Interval
    | Ident
Number_set ::=
    "Z"
    | "N"
    | "N1"
    | "NAT"
    | "NAT1"
    | "INT"

```


B.2 Grammar of B Types

```
Type ::= Basic_type
|      "ℙ" "(" Type ")"
|      Type "x" Type
|      "struct" "(" (Ident ":" Type)+ ")"
|      "(" Type ")"

Basic_type ::=
"Z"
|      "BOOL"
|      "STRING"
|      Ident
```


ANNEX C. VISIBILITY TABLES

Visibility rules between a component C1 and a component C2 define for each constituent of C2 the access modes applied in the clauses of C1. As far as the data is concerned, a distinction is made between read-only access and read-write access. For operations, we distinguish access to consultation operations (operations whose specification does not modify the machine variables) and the modification operations.

In the visibility tables below, MA refers to an abstract machine, M_{n-1} refers to a refinement, M_n refers to a refinement or an implantation, and MB refers to an abstract machine linked to a component by one of the visibility clauses: IMPORTS, SEES, INCLUDES, or USES.

The table below indicates the different modes of visibility of the constituents of the clauses.

Visibility mode	Description
	constituent is not visible
read	visible constituent
read - write	visible constituent, if the constituent is a variable used in a substitution, it can be modified, if the constituent is an operation called in a substitution, the operation can modify the variables of its abstract machine
read - non-write	visible constituent, if the constituent is a variable used in a substitution, it can not be modified, if the constituent is an operation called in a substitution, this operation does not modify the variables of its abstract machine

C.1 Abstract machine MA / Itself

Clauses of MA Constituents of MA	CONSTRAINTS	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS
Formal parameters	read	read		read	read
Sets, enumerated set elements, concrete constants		read	read	read	read
Abstract constants		read	read	read	read
Concrete variables				read	read – write
Non homonymous abstract variables				read	read – write
Developed operations (non promoted)					

C.2 Abstract Machine or Refinement MA / SEES MB Clause

Clauses of MA Constituents of MB	CONSTRAINTS	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS
Formal parameters					
Sets, enumerated set elements, concrete constants		read	read	read	read
Abstract constants		read	read	read	read
Concrete variables					read – non write
Abstract variables					read – non write
Operations					read – non write

C.3 Abstract Machine or Refinement MA / INCLUDES Abstract Machine or Refinement MB Clause

Clauses of MA Constituents of MB	CONSTRAINTS	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS
Formal parameters					
Sets, enumerated set element, concrete constants			read	read	read
Abstract constants			read	read	read
Concrete variables				read	read – non write
Abstract variables				read	read – non write
Operations					read write

C.4 Abstract Machine MA / USES MB Clause

Clauses of MA Constituents of MB	CONSTRAINTS	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS
Formal parameters				read	read
Sets, enumerated set elements, concrete constants			read	read	read
Abstract constants			read	read	read
Concrete variables				read	read – non write
Abstract variables				read	read – non write
Operations					

C.5 Refinement MN / Itself

Clauses of MN Constituents of MN	Parameters of INCLUDES / EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS
Formal parameters	read		read	read
Sets, enumerated set element, concrete constants	read	read	read	read
Abstract constants	read	read	read	read
Concrete variables			read	read - write
Abstract variables			read	read - write
Developed operations				

C.6 Refinement MN / Abstraction MN-1

Clauses of MN Constituents of MN-1	Parameters of INCLUDES /EXTENDS	PROPERTIES	INVARIANT / ASSERTION	INITIALISATION /	OPERATIONS
				Substitutions	Predicates of ASSERT
Abstract constants that disappear in Mn		read	read		read
Abstract variables that disappear in Mn			read		read

C.7 Implementation MN / Itself

Clauses of MN Constituents of MN	Parameters of IMPORTS /EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS		LOCAL_ OPERATIONS
					Instructions	loops variants and invariants ASSERT predicates	
Formal parameters	read			read	read	read	read
Enumerated sets, elements of enumerated sets	read	read	read	read	read	read	read
Deferred sets, concrete constants	read	read	read write	read	read	read	read
Concrete variables				read	read - write	read	read-write
Developed operations							
Local operations					read-write in OPERATIONS but not in INITIALISATION		

C.8 Implementation MN / Abstraction MN-1

Clauses of MN Constituents of MN-1	Parameters of	PROPERTIES	INVARIANT /	INITIALISATION / OPERATIONS		LOCAL_	
	IMPORTS /EXTENDS		ASSERTION	Instructions	loops variants and invariants ASSERT predicates	Substitutions	ASSERT Predicates
Abstract constants		Read	read		read		read
Abstract variables			read		read		read

C.9 Implementation MN / SEES MB Clause

Clauses of MN Constituents of MB	Parameters of IMPORTS / EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS		LOCAL_ OPERATIONS
					Instructions	loops variants and invariants ASSERT predicates	
Formal parameters							
Sets, enumerated set elements, concrete constants	read	read	read	Read	read	read	read
Abstract constants		read		Read		read	read
Concrete variables					read – non write	read	read-non write
Abstract variables						read	read-non write
Operations					read – non write		read-non write

C.10 Implementation MN / IMPORTS MB Clause

Clauses of MN Constituents of MB	Parameters of IMPORTS /EXTENDS	PROPERTIES	VALUES	INVARIANT / ASSERTION	INITIALISATION / OPERATIONS		LOCAL_ OPERATIONS
					Instructions	loops variants and invariants ASSERT predicates	
Formal parameters							
Sets, enumerated set elements, concrete constants		read	read	read	read	read	read
Abstract constants		read		read		read	read
Concrete variables				read	read – non write	read	read-write
Abstract variables				read		read	read- write
Operations					read – write		read-write

ANNEX D. GLOSSARY

Abstract constant Constant value data belonging to a component of any type and which may be refined during component refinement.

Abstract machine The specification of a B module. An abstract machine has clauses that are used to declare the abstract machine links, its static part (sets, parameters, constants, variables and their properties) and its dynamic part (initialization of variables and operations on data).

Abstract machine instance An instance created during the specifications phase by an *inclusion*. It forms an abstract data space.

Abstract module Designates a B module with only one abstract machine that is not refined and that does not have any associated code.

Abstract variable Data of any type belonging to a component and which may be refined during the component refining process.

Abstraction Concept symmetrical to refinement. If component M_n is a refinement of component M_{n-1} , then M_{n-1} is an abstraction of M_n .

Array A total function of a set, or a Cartesian product of sets (if the array is multi-dimensional), towards a set.

Base machine Designates the abstract machine of a base module.

Base module	Designates a B module with only one abstract machine that is not refined and which has manually associated code, that directly implements the machine data and services.
B0	The part of B language used to describe implementation operations and data. B0 language is a computer programming language.
Clause	Components are made up of clauses. Each clause is used to declare a specific part of the component.
Component	Indifferently refers to a machine, a refinement or an implementation.
Concrete constant	Constant value data that belongs to a component that represents a scalar, array, non empty finite integer interval integer. A concrete constant is automatically preserved during refinement.
Concrete machine instance	An instance created during the implementation phase by <i>importing</i> . It forms a concrete data space in the computer program assigned to the project.
Concrete variable	Implementable data belonging to a component and preserved during refinement, representing either a scalar or an array.
Condition	An implementable predicate used in the B0 IF and WHILE instructions.
Constant	Designate either a concrete constant or an abstract constant.
Constituent	A constituent refers to everything that can be named in a component. It may be a set, a constant, a variable, a quantified variable, a local variable, a machine parameter, a predicate or an operation.
Consultation access operation	An operation on a component that does not modify the component variables.
Data	A mathematical object that has a name and a value. The type of any B data item must correspond to the types defined in the mathematical library.
Demonstration	Refer to Proof
Developed module	A developed module is a module entirely developed in B language. It comprises an abstract machine, any refinements and its implementation.

Gluing invariant	A specific invariant from the INVARIANT clause, of a component, that expresses a refinement relation between the abstraction variables and the variable of its abstraction.
Implementation	The last refinement of a developed module. The implementation language is B0 language.
Imported machine instance	A machine instance found in the IMPORTS or EXTENDS clause.
Included machine instance	A machine instance found in the INCLUDES or EXTENDS clause.
Initialization	The initialization of a component instance is described in the INITIALISATION clause. It especially allows assigning a value to variables of the component instance.
Instruction	A substitution that is part of B0 language.
Invariant	A predicate that expresses invariant properties about the data of a component. There are two kinds of invariants in B language: the invariant from the INVARIANT clause, about the component data that can be modified and the invariant of the WHILE loop clause, about the data modified in the loop.
Lexem	A character string that belongs to a lexical unit of a language. The result of the lexical analysis phase of a text is a sequence of lexems.
Local operation	An operation local to an implementation: it is specified in the LOCAL_OPERATIONS clause and implemented in the OPERATIONS clause and usable only from inside the implementation.
Machine	Refer to Abstract machine
Machine instance	A copy that uses an abstract machine as its model. An abstract machine instance has a data space that contains the values of data that may be modified in the machine (variables and parameters).
Main machine	A specific machine in a project that serves as input point for running the code of a B project.
Module	A B module enables modeling a sub-system; it forms a part of a B project. The specification of a B module is formalized in B language in an abstract machine. There are three sorts of modules, modules developed by successive refinements of an abstract machine, abstract modules and base modules. Due to

incorrect language usage, there is often confusion between the module and its specification, the abstract machine.

Operation	A service provided by a B module. The operations are the dynamic part of a module.
Proof obligation	A mathematical lemma made up of a list of predicates called assumptions and a predicate called the goal and which must be proved under these assumptions.
Promoted operation	Operation on a component whose parameters and specifications are identical to an operation on an <i>included</i> or <i>imported</i> machine instance.
Parameter	Three constituents of B language use parameters: abstract machines, definitions and operations. Formal parameters are names given during the declaration of a constituent with parameters. When using a constituent, its parameters are assigned values called the effective parameters.
Proof	A mathematical activity that consists in demonstrating Proof Obligations. Project development mainly comprises two major activities: writing components and demonstrating the proof obligations related to these components.
Project	Designates a complete and self-sufficient set of modules used to formally specifies a system and possibly to generate a computer program that conforms to the formal specifications.
Refinement	The refinement M_n of a component M_{n-1} is a new formulation of M_n , in which certain components of M_n are refined (the abstract constants and the abstract variables, the initialization and the operations).
Refining	Refining a component with certain properties is equivalent to providing a new formulation for this component using one or more new components that must not contradict the properties of the refined component, and reduce the level of component abstraction and indeterminism. Refining also allows enriching a component by adding new specification details not included in the abstraction.
Renaming	The renaming function in B is used to create abstract machine instances. A machine instance is designated by the name of the machine preceded by a renaming prefix. The renaming prefix comprises an identifier followed by a dot. The variables and the operations from a renamed machine instance are designated from the outside using the same renaming prefix.

Required machine	A machine that is <i>seen, included, used, imported</i> or refined.
Seen machine instance	A machine instance found in the SEES clause.
Signature of an operation	An ordered list of input and output parameter types for an operation.
Substitution	A mathematical notation used to model the transformation of predicates.
Typing	A mechanism that verifies statically the type of data.
Used machine instance	A machine instance found in the USES clause of a machine.
Valuation	A mechanism that assigns values to concrete constants and to deferred sets declared in a B module, within the module implementation. The valuing is described in the VALUES clause.
Variable	Designates either a concrete variable or an abstract variable.
Visibility clause	A set of clauses used to link a component with machine instances. The visibility clauses number five: IMPORTS, SEES, INCLUDES, USES and EXTENDS.

#

-, 40, 50
 !, 31
 &, 30
 (, 30
), 30
 =>, 30
 <=>, 30
 #, 31
 =, 32
 /=, 32
 /., 33
 <., 34
 <<., 34
 /<., 34
 /<<., 34
 <=, 35
 <, 35
 >=, 35
 >, 35
 +, 40
 *, 40
 /, 40
 **, 40
 |->, 45
 {}, 46
 *, 48
 {, 48
 }, 48
 ... 48
 V, 50
 Λ, 50
 ', 53
 <->, 55
 -1, 56
 ;, 56
 ><, 56
 ||, 56
 [, 60
], 60
 <|, 61
 <<|, 61
 |>, 61
 |>>, 61
 <+, 61
 +->, 63
 -->, 63
 >+>, 63
 >->, 63
 +->>, 63
 -->>, 63
 >->>, 63
 %, 65
 [], 67
 [, 67
], 67
 ^, 71
 ->, 71
 <-, 71
 //\, 71
 \/, 71
 <->, 169

\$

\$. 38
 \$0, 38

.

., 38

|

||, 109

A

abstract machine, 181, 215
 ABSTRACT_CONSTANTS, 140
 ABSTRACT_VARIABLES, 152
 abstraction, 215
 addition, 40
 analysis
 lexical, 3
 semantic, 3
 syntactic, 3
 anticollision, 3
 ANY, 97
architecture, 181
 arity, 78
 array, 215
 array, 17
 concrete, 17
 ASSERT, 91
 ASSERTIONS, 158

B

B0, 215
 BEGIN, 86
 belonging, 33
 bijection
 total, 63
 bin, 80
 Bool, 39, 144, 197
 BOOL, 46
 Boolean, 16, 17
 bracket, 38
 brackets, 30
 btree, 73

C

card, 43
 cardinal, 43
 Cartesian product, 48
 CASE, 96
character strings, 6
 CHOICE, 92
 clause, 215
 visibility, 219
 closure
 transitive, 59
 transitive and reflexive, 59
comments, 6

component, 111, 215
 composition, 56
 conc, 71
 CONCRETE_CONSTANTS, 138
 CONCRETE_VARIABLES, 150
 condition, 216
 conjunction, 30
 const, 75
 constant, 142, 216
 abstract, 140, 215
 concrete, 20, 138, 144, 215
 gluing, 143
 typing, 142
 CONSTANTS, 138, 197
 constituent, 216
 CONSTRAINTS, 118
 couple, 45

D

data, 38, 216
 definition, 8
définition
 call, 10
 DEFINITIONS, 8
 demonstration, 216
determinism, 84
 development, 181
 difference, 50
 Difference, 40
 disjunction, 30
 DO, 107
 dom, 60
 domain, 60

E

EITHER, 96
 ELSE, 93, 95, 96
 ELSIF, 93
 END, 86, 90, 91, 92, 93, 95, 96, 102, 107, 111, 114, 116,
 196
 equal to, 32
 equivalence, 30
 expression, 37
Expression
 Arithmetical, 40, 43
 Boolean, 39
 Cartesian, 45
 Sets, 46
 extends, 133
 EXTENDS, 133

F

FALSE, 39
 father, 78
 FIN, 48
 FIN1, 48
 first, 69
 fnc, 65
 front, 69
 function
 evaluation, 65
 partial, 63

total, 63
 transformed into, 65

G

greater than or equal to, 35
grouping, 128

H

homonym
 concrete constant, 138
 invariant, 154
 homonymy, 128
 abstract constant, 140
 abstract variable, 152
 concrete constant, 150, 152
 initialization, 159
 properties of constants, 142
 valuation of constants, 144

I

id, 56
 identifier anti-collision, 177
identifiers, 5
 identity, 56
 IF, 93
 image, 60
 implementation, 182, 216
 IMPLEMENTATION, 116
 implication, 30
 import, 120
 IMPORTS, 183
 IN, 102
 INCLUDES, 127, 184
 inclusion
 strict, 34
 inclusion, 34
 indeterminism, 167
 infix, 80
 INITIALISATION, 159
 initialization, 216
 injection
 partial, 63
 total, 63
 instance, 183
 machine, 217
 machine imported, 216
 machine included, 216
 machine seen, 218
 machine used, 218
 instruction, 217
 INT, 46
 integer
 concrete, 16
 INTEGER, 46
 integer division, 40
 inter, 50
 INTER, 50
 intersection, 50
 generalized, 50
 quantified, 50
 interval, 48
 invariant, 154, 217

gluing, 216
 INVARIANT, 107, 154
 iseq, 67
 iseq1, 67
 iteration, 59

K

keywords, 5

L

lambda-expression, 65
 last, 69
 left, 80
 less than or equal, 35
 lexem, 3, 217
 library, 186
link, 183
 rules, 185
 literal
 listed, 136
literal integers, 6
 LOCAL_OPERATIONS, 169

M

machine, 181, 217
 base, 215
 base, 182
 base, 215
 main, 217
 required, 218
 MACHINE, 111
 maplet, 45
 max, 43
 maximum, 43
 MAXINT, 40
 min, 43
 minimum, 43
 MININT, 40
 mirror, 75
 mod, 40
 module
 developed, 182
 module, 181
module
 base, 182
module
 base, 182
 module
 abstract, 182
 module
 abstract, 215
 module
 developed, 216
 module, 217
 modulo, 40

N

NAT, 46
 NAT1_, 46
 NATURAL, 46
 NATURAL1, 46

negation, 30
 non belonging, 33
 non inclusion
 strict, 34
 non inclusion, 34
 not, 30

O

OF, 96
 operation, 163, 217
 body, 165, 167
 consultation, 216
 header, 164
 local, 217
 promoted, 217
 refining, 166
 operation, 169
 OPERATIONS, 163
 or, 30
 OR, 92, 96
 ordered pair, 14, 45
 overwrite, 61

P

parameter, 217
 of machine, 26
 of operations, 164
 operation input, 25, 164
 operation output, 165
 parameters
 machine, 118
 perm, 67
 permutations, 67
 PI, 43
 postfix, 75
 POW, 48
 POW1, 48
 power of, 40
 power-set, 48
 PRE, 90
 precondition, 167
 pred, 40
 predecessor, 40
 predicate
 typing, 15
 predicate, 29
 prefix, 75
 prj1, 56
 prj2, 56
 product, 40
 direct, 56
 of expressions, 43
 parallel, 56
 project, 183, 218
 projection
 first, 56
 second, 56
 promote, 131
 PROMOTES, 131
promotion, 129
 proof, 218
 proof obligation, 217
 PROPERTIES, 142

propositions, 30

Q

quantifier
 existential, 31
 universal, 31

R

ran, 60
 range, 60
 rank, 78
 rec, 53
 record
 field access, 53
 in extension, 53
 set, 53
 record, 18
 refinement, 182, 218
 REFINEMENT, 114
 REFINES, 119
 refining, 218
 rel, 65
 relation
 transformed into, 65
renaming
 SEES, 123
 renaming, 38
renaming, 183
 renaming, 218
 restriction
 domain, 61
 range, 61
 sémantic, 3
 rev, 69
 reverse, 56
 right, 80

S

scope, 3
 sees, 123
 SEES, 184
 SELECT, 95
 seq, 67
 seq1, 67
 sequence
 empty, 67
 first element, 69
 front, 69
 general concatenation, 71
 in extension, 67
 insert at tail, 71
 insert in front, 71
 last element, 69
 restrict at tail, 71
 restrict in front, 71
 reverse, 69
 size, 69
 tail, 69
 sequences, 67
 bijective, 68
 injective, 67
 injective non empty, 67

non empty, 67
 set
 abstract, 136, 144
 empty, 46
 in comprehension, 48
 in extension, 48
 listed, 136
 of Boolean values, 46
 of character strings, 46
 of integers, 46
 of non null integers, 46
 of relative integers, 46
 of subsets, 14
 power-set, 14
 set
 abstract, 16
 set
 enumerated, 16
 Set
 Relations, 55
 SETS, 136
 SIGMA, 43
 size, 69
 sizet, 75
 skip, 87
 son, 78
 sons, 75
spacing characters, 6
 strictly greater than, 35
 strictly less than, 35
 STRING, 46
 struct, 53
 sub-set
 non empty, 48
 sub-sets, 48
 finite, 48
 non empty finite, 48
 substitution, 83, 218
 assertion, 91
 becomes element of, 99
 becomes equal, 89
 becomes such that, 100
 bounded choice, 92
 concrete, 167
 generalized, 84
 identity, 87
 operation call, 105
 precondition, 90
 sequencing, 103
 simultaneous, 109
 VAR, 102
 while, 107
 subtraction
 domain, 61
 range, 61
 subtree, 78
 succ, 40
 successor, 40
 sum
 of expressions, 43
 surjection
 partial, 63
 total, 63
 syntax, 7

T

tail, 69
 THEN, 90, 91, 93, 95, 96, 206
 top, 75
 tree
 arity, 78
 binary in extension, 80
 infix flattening, 80
 left subtree, 80
 rank, 78
 right subtree, 80
 subtree, 78
 tree set, 73
 tree, 73
 tree
 binary tree set, 73
 tree
 construction, 75
 tree
 root, 75
 tree
 sons, 75
 tree
 prefixed flattening, 75
 tree
 postfixed flattening, 75
 tree
 size, 75
 tree
 symmetry, 75
 tree
 father, 78
 tree
 sons, 78
 TRUE, 39

type
 basic, 14
 typing, 3, 12

U

unary minus, 40
 unequal to, 32
 union, 50
 generalized, 50
 quantified, 50
 UNION, 50
 USES, 134, 185
 using, 134

V

valuation, 144, 154, 219
 VALUES, 144
 VAR, 102
 variable, 219
 abstract, 152, 215
 concrete, 150, 216
 initialize, 159
 linkage, 155
 typing, 154
 VARIABLES, 152, 197
 VARIANT, 107
 visibility, 3

W

WHEN, 95
 WHERE, 97
 WHILE, 107