INTERNATIONAL STANDARD

ISO 10303-50

First edition
2002-05-01

Industrial automation systems and integration — Product data representation and exchange —

Part 50:
Integrated generic resource: Mathematical constructs

Systèmes d'automatisation industrielle et intégration — Représentation et échange de données de produits —

Partie 50: Ressources génériques intégrées: Constructions mathématiques

Reference number
ISO 10303-50:2002(E)

© ISO 2002

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO 2002

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

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

Printed in Switzerland

Contents

Page

1 Scope 1

2 Normative references 1

3 Terms, definitions, and symbols 2
3.1 Terms defined in ISO 10303-1 2
3.2 Other terms and definitions 2
3.3 Symbols 5

4 Mathematical functions 6
4.1 Introduction 8
4.2 Fundamental concepts and assumptions 8
4.2.1 Mathematical function 8
4.2.2 Mathematical object or value 9
4.2.3 Mathematical expression 10
4.2.4 Mathematical space 10
4.2.5 Mathematical tuple 11
4.2.6 Function domains and ranges 11
4.2.7 Spaces of one-tuples 12
4.2.8 Array function 12
4.2.9 Table function 12
4.2.10 Matrix 12
4.2.11 Inputs and Parameters 13
4.2.12 Function evaluation 14
4.2.13 Function application 14
4.3 Mathematical functions schema constant definitions 14
4.3.1 schema_prefix 15
4.3.2 the elementary spaces 15
4.3.3 the empty space 15
4.3.4 real intervals 15
4.3.5 tuple spaces 16
4.3.6 empty values 16
4.4 Mathematical functions schema type definitions 17
4.4.1 nonnegative_integer 17
4.4.2 positive_integer 17
4.4.3 zero_or_one 17
4.4.4 one_or_two 18
4.4.5 local names for simple types 18
4.4.6 maths_simple_atom 18
4.4.7 maths_atom 19
4.4.8 atom_based_tuple 19
4.4.9 atom_based_value 19
4.4.10 maths_tuple 20
4.4.11 maths_value 20
4.4.12 maths_expression 20
4.4.13 maths_function_select 21
4.4.14 input_selector 21
4.4.15 elementary_space_enumerators 22
4.4.16 ordering_type 23
4.4.17 lower_upper 24
4.4.18 symmetry_type 24

© ISO 2002 — All rights reserved

4.4.19 elementary_function_enumerators . . . 25
4.4.20 open_closed . . . 31
4.4.21 space_constraint_type . . . 31
4.4.22 repackage_options . . . 32
4.4.23 extension_options . . . 32
4.4.24 maths_enum_atom . . . 33
4.4.25 dotted_express_identifier . . . 34
4.4.26 express_identifier . . . 34
4.4.27 product_space . . . 34
4.4.28 tuple_space . . . 35
4.4.29 maths_space_or_function . . . 35
4.4.30 real_interval . . . 35
4.5 Mathematical functions schema entity definitions . . . 36
4.5.1 quantifier_expression . . . 36
4.5.2 dependent_variable_definition . . . 37
4.5.3 bound_variable_semantics . . . 37
4.5.4 free_variable_semantics . . . 38
4.5.5 complex_number_literal . . . 38
4.5.6 logical_literal . . . 39
4.5.7 binary_literal . . . 39
4.5.8 maths_enum_literal . . . 39
4.5.9 real_tuple_literal . . . 40
4.5.10 integer_tuple_literal . . . 40
4.5.11 atom_based_literal . . . 40
4.5.12 maths_tuple_literal . . . 41
4.5.13 maths_variable . . . 41
4.5.14 maths_real_variable . . . 42
4.5.15 maths_integer_variable . . . 42
4.5.16 maths_boolean_variable . . . 43
4.5.17 maths_string_variable . . . 43
4.5.18 function_application . . . 44
4.5.19 maths_space . . . 45
4.5.20 elementary_space . . . 46
4.5.21 finite_integer_interval . . . 46
4.5.22 integer_interval_from_min . . . 47
4.5.23 integer_interval_to_max . . . 47
4.5.24 finite_real_interval . . . 47
4.5.25 real_interval_from_min . . . 48
4.5.26 real_interval_to_max . . . 49
4.5.27 cartesian_complex_number_region . . . 49
4.5.28 polar_complex_number_region . . . 50
4.5.29 finite_space . . . 51
4.5.30 uniform_product_space . . . 52
4.5.31 listed_product_space . . . 53
4.5.32 extended_tuple_space . . . 54
4.5.33 function_space . . . 55
4.5.34 maths_function . . . 56
4.5.35 finite_function . . . 57
4.5.36 constant_function . . . 58
4.5.37 selector_function . . . 59
4.5.38 elementary_function . . . 60
4.5.39 restriction_function . . . 60

 © ISO 2002 — All rights reserved

4.5.40 repackaging_function . . . 61
4.5.41 reindexed_array_function . . . 63
4.5.42 series_composed_function . . . 64
4.5.43 parallel_composed_function . . . 64
4.5.44 explicit_table_function . . . 67
4.5.45 listed_real_data . . . 68
4.5.46 listed_integer_data . . . 69
4.5.47 listed_logical_data . . . 69
4.5.48 listed_string_data . . . 70
4.5.49 listed_complex_number_data . . . 71
4.5.50 listed_data . . . 72
4.5.51 externally_listed_data . . . 72
4.5.52 linearized_table_function . . . 73
4.5.53 standard_table_function . . . 75
4.5.54 regular_table_function . . . 76
4.5.55 triangular_matrix . . . 78
4.5.56 strict_triangular_matrix . . . 78
4.5.57 symmetric_matrix . . . 79
4.5.58 symmetric_banded_matrix . . . 80
4.5.59 banded_matrix . . . 81
4.5.60 basic_sparse_matrix . . . 83
4.5.61 homogeneous_linear_function . . . 85
4.5.62 general_linear_function . . . 86
4.5.63 b_spline_basis . . . 87
4.5.64 b_spline_function . . . 88
4.5.65 rationalize_function . . . 89
4.5.66 partial_derivative_function . . . 91
4.5.67 partial_derivative_expression . . . 92
4.5.68 definite_integral_function . . . 94
4.5.69 definite_integral_expression . . . 95
4.5.70 abstracted_expression_function . . . 97
4.5.71 expression_denoted_function . . . 98
4.5.72 imported_point_function . . . 99
4.5.73 imported_curve_function . . . 100
4.5.74 imported_surface_function . . . 100
4.5.75 imported_volume_function . . . 101
4.5.76 application_defined_function . . . 102
4.5.77 mathematical_description . . . 103
4.6 Mathematical functions schema function definitions . . . 104
4.6.1 all_members_of_es . . . 104
4.6.2 any_space_satisfies . . . 106
4.6.3 assoc_product_space . . . 107
4.6.4 atan2 . . . 109
4.6.5 bool . . . 110
4.6.6 check_sparse_index_domain . . . 110
4.6.7 check_sparse_loc_range . . . 111
4.6.8 check_sparse_index_to_loc . . . 112
4.6.9 compare_basis_and_coef . . . 113
4.6.10 compare_list_and_value . . . 113
4.6.11 compare_values . . . 114
4.6.12 compatible_complex_number_regions . . . 115
4.6.13 compatible_es_values . . . 117

© ISO 2002 — All rights reserved

4.6.14 compatible_intervals . . . 118
4.6.15 compatible_spaces . . . 118
4.6.16 composable_sequence . . . 124
4.6.17 convert_to_literal . . . 124
4.6.18 convert_to_maths_function . . . 125
4.6.19 convert_to_maths_value . . . 125
4.6.20 convert_to_operand . . . 126
4.6.21 convert_to_operands . . . 127
4.6.22 convert_to_operands_prcmfn . . . 128
4.6.23 definite_integral_check . . . 128
4.6.24 definite_integral_expr_check . . . 129
4.6.25 derive_definite_integral_domain . . . 130
4.6.26 derive_elementary_function_domain . . . 132
4.6.27 derive_elementary_function_range . . . 135
4.6.28 derive_finite_function_domain . . . 137
4.6.29 derive_finite_function_range . . . 138
4.6.30 derive_function_domain . . . 138
4.6.31 derive_function_range . . . 141
4.6.32 domain_from . . . 144
4.6.33 dot_count . . . 145
4.6.34 dotted_identifiers_syntax . . . 145
4.6.35 drop_numeric_constraints . . . 146
4.6.36 enclose_cregion_in_pregion . . . 147
4.6.37 enclose_pregion_in_cregion . . . 151
4.6.38 enclose_pregion_in_pregion . . . 154
4.6.39 equal_cregion_pregion . . . 160
4.6.40 equal_maths_functions . . . 162
4.6.41 equal_maths_spaces . . . 163
4.6.42 equal_maths_values . . . 166
4.6.43 es_subspace_of_es . . . 168
4.6.44 expression_is_constant . . . 169
4.6.45 extract_factors . . . 169
4.6.46 extremal_position_check . . . 170
4.6.47 factor1 . . . 171
4.6.48 factor_space . . . 172
4.6.49 free_variables_of . . . 172
4.6.50 function_applicability . . . 173
4.6.51 function_is_1d_array . . . 174
4.6.52 function_is_1d_table . . . 175
4.6.53 function_is_2d_table . . . 176
4.6.54 function_is_array . . . 177
4.6.55 function_is_table . . . 177
4.6.56 has_values_space . . . 178
4.6.57 list_selected_components . . . 180
4.6.58 make_abstracted_expression_function . . . 180
4.6.59 make_atom_based_literal . . . 181
4.6.60 make_b_spline_basis . . . 181
4.6.61 make_b_spline_function . . . 182
4.6.62 make_banded_matrix . . . 182
4.6.63 make_basic_sparse_matrix . . . 183
4.6.64 make_binary_literal . . . 184
4.6.65 make_boolean_literal . . . 184

 © ISO 2002 — All rights reserved

4.6.66 make_cartesian_complex_number_region . . . 185
4.6.67 make_complex_number_literal . . . 185
4.6.68 make_constant_function . . . 186
4.6.69 make_cos_expression . . . 186
4.6.70 make_definite_integral_expression . . . 187
4.6.71 make_definite_integral_function . . . 187
4.6.72 make_elementary_function . . . 188
4.6.73 make_elementary_space . . . 188
4.6.74 make_environment . . . 189
4.6.75 make_expression_denoted_function . . . 189
4.6.76 make_extended_tuple_space . . . 190
4.6.77 make_finite_function . . . 190
4.6.78 make_finite_integer_interval . . . 191
4.6.79 make_finite_real_interval . . . 191
4.6.80 make_finite_space . . . 192
4.6.81 make_function_application . . . 192
4.6.82 make_function_space . . . 193
4.6.83 make_general_linear_function . . . 194
4.6.84 make_int_literal . . . 194
4.6.85 make_integer_interval_from_min . . . 195
4.6.86 make_listed_complex_number_data . . . 195
4.6.87 make_listed_data . . . 196
4.6.88 make_listed_integer_data . . . 196
4.6.89 make_listed_product_space . . . 197
4.6.90 make_listed_real_data . . . 197
4.6.91 make_logical_literal . . . 198
4.6.92 make_maths_enum_literal . . . 198
4.6.93 make_maths_real_variable . . . 199
4.6.94 make_maths_tuple_literal . . . 199
4.6.95 make_mult_expression . . . 200
4.6.96 make_parallel_composed_function . . . 200
4.6.97 make_partial_derivative_expression . . . 201
4.6.98 make_partial_derivative_function . . . 201
4.6.99 make_polar_complex_number_region . . . 202
4.6.100 make_rationalize_function . . . 202
4.6.101 make_real_interval_from_min . . . 203
4.6.102 make_real_interval_to_max . . . 203
4.6.103 make_real_literal . . . 204
4.6.104 make_regular_table_function . . . 204
4.6.105 make_reindexed_array_function . . . 205
4.6.106 make_repackaging_function . . . 205
4.6.107 make_selector_function . . . 206
4.6.108 make_series_composed_function . . . 207
4.6.109 make_sin_expression . . . 207
4.6.110 make_standard_table_function . . . 208
4.6.111 make_strict_triangular_matrix . . . 208
4.6.112 make_string_literal . . . 209
4.6.113 make_unary_minus_expression . . . 210
4.6.114 make_uniform_product_space . . . 210
4.6.115 max_exists . . . 211
4.6.116 max_included . . . 211
4.6.117 member_of . . . 212

© ISO 2002 — All rights reserved

4.6.118 min_exists . . . 217
4.6.119 min_included . . . 217
4.6.120 no_cyclic_domain_reference . . . 218
4.6.121 no_cyclic_space_reference . . . 219
4.6.122 nondecreasing . . . 220
4.6.123 number_superspace_of . . . 220
4.6.124 number_tuple_subspace_check . . . 221
4.6.125 one_tuples_of . . . 221
4.6.126 parallel_composed_function_composability_check . . . 222
4.6.127 parallel_composed_function_domain_check . . . 222
4.6.128 parse_express_identifier . . . 223
4.6.129 partial_derivative_check . . . 224
4.6.130 real_max . . . 225
4.6.131 real_min . . . 225
4.6.132 regular_indexing . . . 226
4.6.133 remove_first . . . 227
4.6.134 repackage . . . 227
4.6.135 shape_of_array . . . 228
4.6.136 simplify_function_application . . . 229
4.6.137 simplify_generic_expression . . . 243
4.6.138 simplify_maths_space . . . 249
4.6.139 simplify_maths_value . . . 250
4.6.140 singleton_member_of . . . 251
4.6.141 space_dimension . . . 252
4.6.142 space_is_continuum . . . 252
4.6.143 space_is_singleton . . . 253
4.6.144 stripped_typeof . . . 254
4.6.145 subspace_of . . . 254
4.6.146 subspace_of_es . . . 260
4.6.147 substitute . . . 261
4.6.148 values_space_of . . . 263

Annex A (normative) Short names of entities . . . 266

Annex B (normative) Information object registration . . . 269

Annex C (informative) Computer-interpretable listings . . . 270

Annex D (informative) EXPRESS-G diagrams . . . 271

Bibliography . . . 282

Index . . . 283

Figures

Figure 1 — Schema relationships of the mathematical_functions_schema . . . xi
Figure D.1 — EXPRESS-G diagram of the mathematical_functions_schema (1 of 10) . . . 272
Figure D.2 — EXPRESS-G diagram of the mathematical_functions_schema (2 of 10) . . . 273
Figure D.3 — EXPRESS-G diagram of the mathematical_functions_schema (3 of 10) . . . 274
Figure D.4 — EXPRESS-G diagram of the mathematical_functions_schema (4 of 10) . . . 275
Figure D.5 — EXPRESS-G diagram of the mathematical_functions_schema (5 of 10) . . . 276
Figure D.6 — EXPRESS-G diagram of the mathematical_functions_schema (6 of 10) . . . 277

 © ISO 2002 — All rights reserved

Figure D.7 — EXPRESS-G diagram of the mathematical_functions_schema (7 of 10) . . . 278
Figure D.8 — EXPRESS-G diagram of the mathematical_functions_schema (8 of 10) . . . 279
Figure D.9 — EXPRESS-G diagram of the mathematical_functions_schema (9 of 10) . . . 280
Figure D.10 — EXPRESS-G diagram of the mathematical_functions_schema (10 of 10) . . 281

Tables

Table 1 — Mathematical symbology 6
Table 2 — Orderings indicated by ordering_type 23
Table A.1 — Short names of entities 266

© ISO 2002 — All rights reserved

Foreword

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

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

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

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

ISO 10303–50 was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC4, *Industrial data.*

This International Standard is organized as a series of parts, each published separately. The structure of this International Standard is described in ISO 10303-1.

Each part of this International Standard is a member of one of the following series: description methods, implementation methods, conformance testing methodology and framework, integrated generic resources, integrated application resources, application protocols, abstract test suites, application interpreted constructs, and application modules. This part is a member of the integrated generic resources series. The integrated generic resources and the integrated application resources specify a single conceptual product data model.

A complete list of parts of ISO 10303 is available from the Internet:

`<http://www.nist.gov/sc4/editing/step/titles/>`

Annexes A and B form a normative part of this part of ISO 10303. Annexes C and D are for information only.

 © ISO 2002 — All rights reserved

Introduction

ISO 10303 is an International Standard for the computer-interpretable representation of product information and for the exchange of product data. The objective is to provide a neutral mechanism capable of describing products throughout their life cycle. This mechanism is suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases, and as a basis for archiving.

This part of ISO 10303 is a member of the integrated resource series. This part of ISO 10303 specifies the **mathematical_functions_schema**.

This part of ISO 10303 specifies EXPRESS data representations for a large class of mathematical functions, expressions, and arrays. They are intended to be used to communicate product property data and related engineering analysis data. Familiarity with the branches of mathematics commonly used in engineering applications is assumed. The central concept is that mathematical functions and arrays are the abstract data objects of interest for expressing any property which requires more than a single number as a value, for describing deterministic relationships between properties, and for documenting behavioural responses of products to varying conditions.

The relationships of the schema in this part of ISO 10303 to other schemas that define the integrated resources of this International Standard are illustrated in Figure 1 using the EXPRESS-G notation. EXPRESS-G is defined in annex D of ISO 10303-11. The documents containing the specifications for these related schemas are identified in note 1 at the beginning of clause 4.

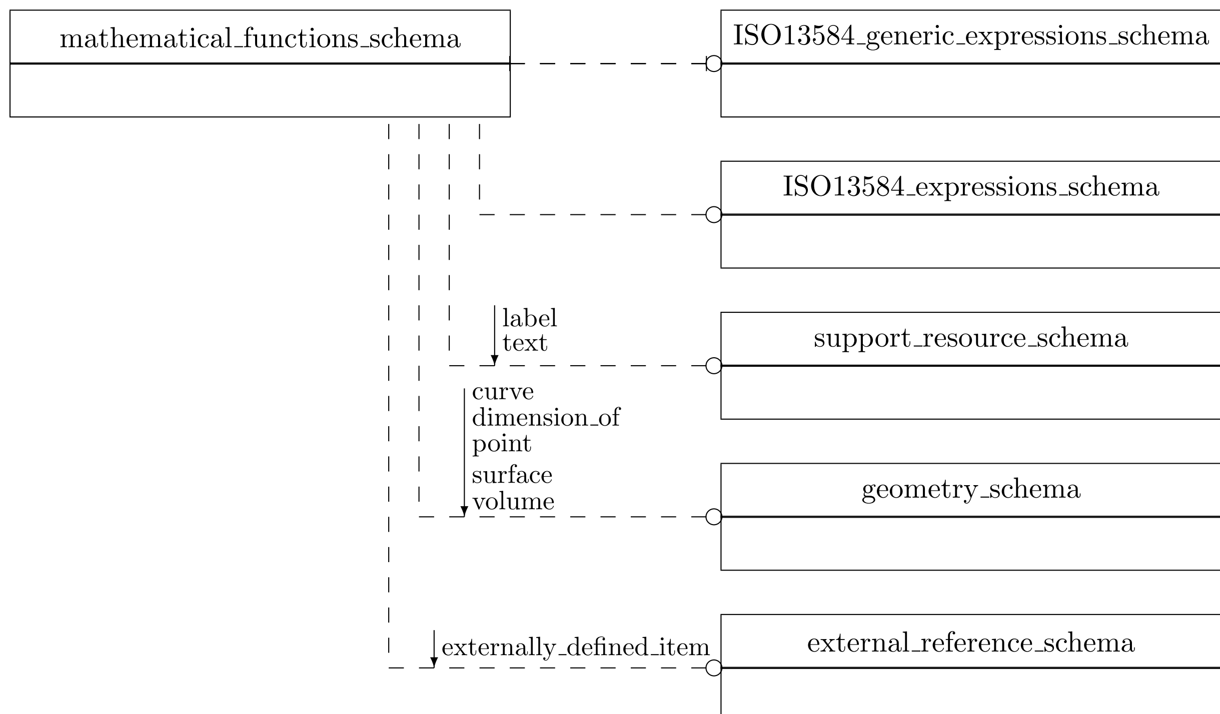

Figure 1 – Schema relationships of the mathematical_functions_schema

© ISO 2002 — All rights reserved

Industrial automation systems and integration — Product data representation and exchange — Part 50 : Integrated generic resource: Mathematical constructs

1 Scope

This part of ISO 10303 specifies the resource constructs for the explicit representation of mathematical structures and data related to properties of a product.

The following are within the scope of this part of ISO 10303:

— multi-dimensional tables;

— mathematical expressions;

— mathematical functions;

— mathematical spaces.

The following are outside the scope of this part of ISO 10303:

— context of application;

— physical units;

— non-mathematical semantics.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO 10303. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 8824-1:1998, *Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation*

ISO 10303-1:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles*

© ISO 2002 — All rights reserved

ISO 10303-11:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual*

ISO 10303-41:2000, *Industrial automation systems and integration — Product data representation and exchange — Part 41: Integrated generic resource: Fundamentals of product description and support*

ISO 10303-42:2000, *Industrial automation systems and integration — Product data representation and exchange — Part 42: Integrated generic resource: Geometric and topological representation*

ISO 13584-20:1998, *Industrial automation systems and integration — Parts Library — Part 20: Logical resource: Logical model of expressions*

3 Terms, definitions, and symbols

3.1 Terms defined in ISO 10303-1

For the purpose of this part of ISO 10303, the following terms defined in ISO 10303-1 apply:

— integrated resource

3.2 Other terms and definitions

For the purposes of this part of ISO 10303, the following definitions apply:

3.2.1
actual function domain
mathematical space containing precisely the tuples of inputs to the function which are related to some tuple of outputs

NOTE See the definitions for "mathematical function" and "function domain". See also 4.2.1 and 4.2.6.

3.2.2
actual function range
mathematical space containing precisely the tuples of outputs from the function which are related to some tuple of inputs

NOTE See the definitions for "mathematical function" and "function range". See also 4.2.1 and 4.2.6.

3.2.3
array function
function whose domain is either a Cartesian product of finite intervals of integers or the one-tuples from such a Cartesian product

NOTE An element of such a domain is a subscript tuple. Evaluation of the function at such an element produces the "array entry" for that subscript tuple.

3.2.4
bound variable
variable which has been specifically referenced by a quantifier in a quantifier expression

 © ISO 2002 — All rights reserved

NOTE A bound variable is no longer available for substitution by a constant value. The semantics of the quantifier expression uses all possible values of the variable rather than one (perhaps undetermined) value.

EXAMPLE 1 The variable x in the statement $\forall x(x = x)$.

EXAMPLE 2 The variable x in the set declaration $\{x|\, x^2 = x\}$.

EXAMPLE 3 The variable x in the definite integral $\int_1^2 \ln(x)\, dx$.

EXAMPLE 4 The variable x in the function definition $f(x) \equiv x + 2$.

3.2.5
Cartesian product space

mathematical space consisting of all ordered tuples whose components are members of the corresponding factor spaces of the product

NOTE Some care must be taken to be explicit about which spaces are the factors entering into the Cartesian product. For example, $R^3 \times R^3$ is the space of ordered pairs of ordered triples from space R, not the space of ordered sextuples from space R. When the factor spaces of the explicit factors are to be treated as the factor spaces of the Cartesian product, that is, the member tuples are to be concatenated rather than entupled, a subscript 'a' (indicating the "associative" product) is appended to the Cartesian product symbol. Thus, $R^3 \times_a R^3 \equiv R^6$.

3.2.6
compatible spaces

mathematical spaces whose intersection is not determined to be empty by the algorithm represented by the EXPRESS function **compatible_spaces**

NOTE Two spaces are compatible if their intersection is not "obviously" empty. In practice, "obvious" is defined by what is detected by the algorithm. The purpose is to rule out some obvious nonsense while permitting ignorance.

3.2.7
computable function

mathematical function for which the relationship between tuples of inputs and tuples of outputs is expressible by means of an algorithm which takes the inputs and produces the related outputs

NOTE All the mathematical functions of interest in the intended applications of this part of ISO 10303 are computable. Nevertheless, the possibility of representing non-computable functions is not ruled out.

3.2.8
expression

language construct composed of constants, variables, operators, quantifiers, and grouping markers, organized in accordance with the language's grammar and denoting some object in the domain of the language

3.2.9
free variable

variable which has not been bound by a quantifier in the expression in which it is used

NOTE A free variable denotes an undetermined member of some set of possible values. There is an implicit assumption that it may be substituted by any constant denoting one of those values.

© ISO 2002 — All rights reserved

3.2.10
function abstraction
quantifier operation which creates an expression denoting a mathematical function from another expression and a list of variables

NOTE The variables become bound and identify the inputs to the function. The algorithm to be used to produce an output is implied by the expression. The "abstraction" comes from the change in the semantics of what the new expression denotes. The expression going into the operation denotes some object. The resulting expression denotes a higher-order object, namely a function which produces objects of the original type from inputs of the types associated with the variables named as inputs.

3.2.11
function application
operation which creates an expression by applying a function to appropriate inputs

NOTE The resulting expression denotes the object which would be the first (and, usually, only) output of the function when it is applied to the given inputs and evaluated.

3.2.12
function domain
mathematical space specifying all nominally valid tuples of inputs to the mathematical function

NOTE See the definitions for "mathematical function" and "actual function domain". See also 4.2.1 and 4.2.6.

3.2.13
function evaluation
process of ascertaining either the outputs a function associates to given inputs, or an error indication if the function does not associate any outputs to the given inputs

NOTE In the case of a computable function, the process amounts to carrying out the algorithm on the inputs to produce the outputs, or an error indication.

3.2.14
function range
mathematical space specifying all permissible tuples of outputs from the mathematical function

NOTE See the definitions for "mathematical function" and "actual function range". See also 4.2.1 and 4.2.6.

3.2.15
mathematical function
relationship between tuples of mathematical objects called inputs and tuples of mathematical objects called outputs in which each tuple of inputs is related to at most one tuple of outputs, together with a mathematical space specifying the function domain and a mathematical space specifying the function range

NOTE The phrase "mathematical function" in this part of ISO 10303 implies a specification of the domain and range in addition to the pairing of inputs and outputs. See 4.2.1 and 4.2.6 for the reasons this definition was chosen for this part of ISO 10303.

 © ISO 2002 — All rights reserved

3.2.16
mathematical object; mathematical value
abstract object from the domain of mathematics

NOTE The terms mathematical object and mathematical value are formally synonymous in this part of ISO 10303. There is a tendency to favor the use of mathematical *value* for simple objects like numbers, characters, and enumeration values, and to favor *object* for complex objects like spaces, functions and expressions, but no formal distinction is being made.

3.2.17
mathematical space; mathematical set
instance of the fundamental aggregate object type in the domain of mathematics

NOTE No topological, metric, ordering, or vector structure is implied or required. In this part of ISO 10303, mathematical space is synonymous with mathematical set. See 4.2.4.

3.2.18
matrix
table function whose domain contains pairs of integers

3.2.19
table function
array function with the additional constraint that all the integer interval factor spaces of the domain either start at zero or at one

NOTE The implication of the constraint is that only the relative values of the subscripts are significant. Hence, one can use pure relative positions (all subscript ranges start from zero), or pure ordinal positions (all subscript ranges start from one), depending on the usual practice in the application area.

3.2.20
tuple
linearly ordered, aggregate mathematical object which has finitely many component mathematical objects

NOTE A tuple contains a first component, a second component, et cetera, up to some maximum which is the length of the tuple. The tuple of zero length exists and has no components.

3.2.21
variable
atomic expression component which has the semantics of denoting an undetermined value from some space of possible values

NOTE See *free* variable and *bound* variable.

3.3 Symbols

For the purposes of this part of ISO 10303, the following symbols and definitions apply.

The mathematical symbol convention used in the mathematical functions schema is given in Table 1.

© ISO 2002 — All rights reserved

Table 1 – Mathematical symbology

Symbol	Definition
$\times$	Cartesian product of spaces
$\times_a$	associative Cartesian product of spaces
$\cup$	set union of spaces
B^m	The Cartesian product of *m* factors of space *B*
R^m	m-dimensional real space
Σ	summation of numerical terms
Π	product of numerical factors

4 Mathematical functions

The following EXPRESS declaration begins the **mathematical_functions_schema** and identifies the necessary external references.

EXPRESS specification:

```
*)
SCHEMA mathematical_functions_schema;
-- Master document: ISO 10303-50:2002
-- EXPRESS last modified: 2001-09-07

REFERENCE FROM ISO13584_generic_expressions_schema     -- ISO 13584-20
  (binary_generic_expression,
   environment,
   generic_expression,
   generic_literal,
   generic_variable,
   multiple_arity_generic_expression,
   simple_generic_expression,
   unary_generic_expression,
   variable_semantics);

REFERENCE FROM ISO13584_expressions_schema             -- ISO 13584-20
  (abs_function                  AS abs_expression,
   acos_function                 AS acos_expression,
   and_expression,
   asin_function                 AS asin_expression,
   atan_function                 AS atan_expression,
   binary_boolean_expression,
   binary_function_call          AS binary_numeric_call_expression,
   binary_numeric_expression,
   boolean_defined_function      AS boolean_defined_expression,
   boolean_expression,
   boolean_literal,
   boolean_variable,
   comparison_equal,
   comparison_expression,
   comparison_greater,
   comparison_greater_equal,
   comparison_less,
   comparison_less_equal,
```

 © ISO 2002 — All rights reserved

```
comparison_not_equal,
concat_expression,
cos_function                   AS cos_expression,
defined_function               AS defined_expression,
div_expression,
equals_expression,
exp_function                   AS exp_expression,
expression,
format_function                AS format_expression,
index_expression,
int_literal,
int_numeric_variable,
int_value_function             AS int_value_expression,
integer_defined_function       AS integer_defined_expression,
interval_expression,
length_function                AS length_expression,
like_expression,
literal_number,
log_function                   AS log_expression,
log10_function                 AS log10_expression,
log2_function                  AS log2_expression,
maximum_function               AS maximum_expression,
minimum_function               AS minimum_expression,
minus_expression,
minus_function                 AS unary_minus_expression,
mod_expression,
mult_expression,
multiple_arity_boolean_expression,
multiple_arity_function_call   AS multiple_arity_numeric_call_expression,
multiple_arity_numeric_expression,
not_expression,
numeric_defined_function       AS numeric_defined_expression,
numeric_expression,
numeric_variable,
odd_function                   AS odd_expression,
or_expression,
plus_expression,
power_expression,
real_defined_function          AS real_defined_expression,
real_literal,
real_numeric_variable,
simple_boolean_expression,
simple_numeric_expression,
simple_string_expression,
sin_function                   AS sin_expression,
slash_expression,
sql_mappable_defined_function  AS sql_mappable_defined_expression,
square_root_function           AS square_root_expression,
string_defined_function        AS string_defined_expression,
string_expression,
string_literal,
string_variable,
substring_expression,
tan_function                   AS tan_expression,
unary_boolean_expression,
unary_function_call            AS unary_numeric_call_expression,
unary_numeric_expression,
value_function                 AS value_expression,
variable,
```

© ISO 2002 — All rights reserved

```
   xor_expression);

REFERENCE FROM support_resource_schema              -- ISO 10303-41
  (label,
   text);

REFERENCE FROM external_reference_schema            -- ISO 10303-41
  (externally_defined_item);

REFERENCE FROM geometry_schema                      -- ISO 10303-42
  (curve,
   dimension_of,
   point,
   surface,
   volume);

(*
```

NOTE 1 The schemas referenced above can be found in the following parts of ISO 10303 and ISO 13584:

ISO13584_generic_expressions_schema	ISO 13584-20
ISO13584_expressions_schema	ISO 13584-20
support_resource_schema	ISO 10303-41
external_reference_schema	ISO 10303-41
geometry_schema	ISO 10303-42

NOTE 2 All entity types defined in the two ISO 13584-20 schemas are explicitly referenced in this schema. The identifiers in the reference statement are arranged in alphabetic order in order to facilitate identification of the defining schema.

NOTE 3 All entity types from **ISO13584_expressions_schema** whose names contained the word "function" have been renamed using the word "expression" in this part of ISO 10303. This change reflects the fact that instances of those types do not represent mathematical functions, but rather the application of certain functions to operands to form mathematical expressions.

NOTE 4 The references to **geometry_schema** are required only for the definition of the **imported_-[point,curve,surface,volume]_function** entity types.

NOTE 5 See annex D, Figures D.1 to D.10, for a graphical presentation of this schema.

4.1 Introduction

The subject of the **mathematical_functions_schema** is the representation of mathematical functions. The context for these mathematical functions is provided by other parts of ISO 10303 and the applications which use this schema. Only the unitless, purely mathematical aspects of mathematical functions are considered in this schema.

4.2 Fundamental concepts and assumptions

4.2.1 Mathematical function

As used in this schema, the phrase mathematical function refers to an identified functional relationship between mathematical objects called inputs and mathematical objects called outputs, together with a specification of the number and kinds of inputs, and a specification of the number and kinds of outputs.

 © ISO 2002 — All rights reserved

A relationship is a functional relationship when the inputs determine the outputs. Usually, a functional relationship exists because the outputs associated with given inputs can be computed from the inputs by means of an algorithm.

In this schema, the specification of the number and kinds of inputs is accomplished by specifying the domain of the function as a mathematical tuple space, where the number of factors of the tuple space indicates the number of individual inputs to the function and the corresponding factor spaces of the tuple space identify the kinds of the inputs.

Analogously, the specification of the number and kinds of outputs is accomplished by specifying the range of the function as a mathematical tuple space, where the number of factors of the tuple space indicates the number of individual outputs from the function and the corresponding factor spaces of the tuple space identify the kinds of the outputs.

NOTE 1 The concept of mathematical function defined in this part of ISO 10303 is a blend of the computer science concept and the mathematical concepts. The exact form it assumes was driven by the requirements of data exchange using the existing structure of ISO 10303.

NOTE 2 Mathematics has two basic views of functions. In the set theoretic view, a function is a set of ordered pairs no two of which have the same first member. In the algorithmic view, a function is a well-formed expression in a formal language with a semantics equivalent to a programming language. Neither of these views is appropriate for the present purpose. The present purpose requires only that sufficient information be recorded so that a function can be reliably reproduced on a different system than that in which it originated. It is assumed that certain very general algorithms are sufficiently well known and understood that it is sufficient to associate them with a named subtype and supply as attributes the parameters necessary to determine a particular instance of that type of function. Examples of such general algorithms are algebraic expression evaluation, matrix multiplication, B-spline evaluation, partial differentiation, and tensor contraction.

NOTE 3 Some of the simplest and most general algorithms are those that combine existing functions to make new functions. Examples are composition of functions (that is, applying a second function to the output of a first function), concatenation of the outputs of two or more functions, and taking linear combinations of functions. In order to identify which functions it makes sense to combine in such ways, functions must be classified according to the kinds of inputs they require and the kinds of outputs they produce. This consideration leads to the inclusion of representations of mathematical spaces, mathematical values and mathematical ordered tuples in the schema.

NOTE 4 Theoretically, the functional relationship precisely determines an actual domain and an actual range and the nominal domain and range required above are redundant. Unfortunately, it is generally impossible to derive this information reliably, as it is equivalent to solving the Turing Machine Halting Problem. Even in simple cases, it is impractical. Consider a function defined as the reciprocal of a fifth degree polynomial. Its actual domain is all real numbers except the one to five real roots of the polynomial. There is no algebraic formula for solving fifth degree (or higher) polynomials. Rather than trying to deal with the complexity of actual domains or ranges, it is far better to specify the nominal domain of such a function as all real numbers, and then accept that the function may fail to produce outputs for some inputs. Similarly, it is better to specify the nominal range of a function and accept that not all values in that range actually appear as outputs.

4.2.2 Mathematical object or value

The phrases “mathematical object” and “mathematical value” are synonymous in this part of ISO 10303. Both are used for any abstract object from the domain of mathematics. As a consequence of providing representations of mathematical functions, there is need to provide a representation of any mathematical object which could be an input to a mathematical function, an output from a mathematical function, or used in a definition of a mathematical function. The EXPRESS language “select” type **maths_value** (see 4.4.11) is defined in this schema

© ISO 2002 — All rights reserved

with the intent of encompassing all EXPRESS language representations of mathematical values.

NOTE 1 The **maths_value** type is highly recursive and includes values isomorphic to any instance or value constructed using this schema except expressions containing free variables and instances of **mathematical_description**.

NOTE 2 There is one fundamental philosophical mismatch between mathematical values and their EXPRESS representations. EXPRESS instances can be created, modified, and destroyed. Mathematical values merely exist. In mathematics there is exactly one object which is the integer three, one object which is the space of all real numbers, and one object which is the (set theoretic) trigonometric sine function taking a real number as input and producing a real number as output. In the context of a language whose constructs represent mathematical objects, however, there will necessarily be many instances representing the same object (for example, $3, 2+1, 5-2, |-3|, e^{ln3}$), so no attempt is made in this schema to try to prevent this redundancy even at the simple level of multiple instances of the same entity type with the same attribute values. With the notable exception of instances of **maths_variable**, only value equality or inequality has any significance within this schema.

4.2.3 Mathematical expression

A mathematical expression is a linguistic construct, possibly containing free variables, which denotes a mathematical object once any free variables have been substituted by appropriate constants. An expression is composed of symbols representing constants, variables, operators, quantifiers, and grouping marks and is constructed in accordance with a grammar.

NOTE 1 In effect, a mathematical expression is a template with some missing parts indicated by free variables. There is a close relationship between mathematical expressions and mathematical functions. Either can be used to construct instances of the other. The difference is that an expression denotes an undetermined object of a certain sort and a corresponding function denotes a whole functional relationship among objects of that sort.

EXAMPLE If n is an integer variable, then $n+2$ is an expression denoting an (indeterminate) integer, and it makes sense to ask whether it is, for example, an even integer, even though no answer can be given in the absence of further information.

Corresponding to this expression, there is a function, which might be described in English as "add two", which could be expressed in the formalism of the Lambda Calculus (see, for example, [4]) as $\lambda n(n+2)$, where lambda is the quantifier indicating "function abstraction" with respect to the associated variable. The variable n in this latter expression is now a bound variable rather than a free variable. This latter expression has no free variables and therefore denotes a mathematical constant. However, the constant denoted is of a higher order than an integer, namely that of functions from integers to integers. It does not make any sense to ask whether this object is an even integer, since it is not an integer.

NOTE 2 The designation of a specific formal language, grammar, and symbology for mathematics is outside the scope of this part of ISO 10303. It is expected that the correspondence between the EXPRESS representations for mathematical expressions in this part of ISO 10303 and any formal or informal language for mathematics will be clear and unambiguous.

4.2.4 Mathematical space

In this part of ISO 10303, mathematical space is a synonym for mathematical set, the fundamental aggregate of mathematics.

NOTE 1 In many contexts the word "space" implies a set with some additional structure, such as in "topological space", "vector space", "Euclidean space", "projective space", or "Hilbert space". Here it has no such implication. The term "space" is preferred in this part of ISO 10303 to avoid confusion with the EXPRESS concept of set. In actual implementations, the EXPRESS set construct can only be used

 © ISO 2002 — All rights reserved

to represent finite sets. Almost all the mathematical spaces of interest have infinitely many members. Consequently, they cannot be directly represented by an EXPRESS set value.

NOTE 2 The present edition provides representations for only the simplest and most useful mathematical spaces. The addition of further subtypes to **maths_space** awaits specific application needs.

4.2.5 Mathematical tuple

The second most important aggregate in mathematics after mathematical set is the ordered tuple. An ordered tuple has a length, a first component, a second component, et cetera, up to the length of the tuple. The components need not have any common type.

The EXPRESS list aggregate type is chosen to represent mathematical ordered tuples in this schema.

Mathematically, there is a unique zero-tuple object. As with many vacuous or trivial or extremal objects in mathematics it plays a useful simplifying role in some applications. It is represented by the zero-length list and such a list is explicitly included in **maths_tuple**, but will usually be excluded in the related tuple types.

4.2.6 Function domains and ranges

The classification of mathematical functions by the “kinds of inputs they require” and the “kinds of outputs they produce” is accomplished by explicitly representing the mathematical spaces which contain all legitimate inputs and all possible outputs. In mathematical terminology, these spaces are the domains and ranges of the functions, respectively.

In order to talk about *the* domain of a function of several inputs, the several “natural” inputs are implicitly combined into an ordered tuple and the space of all such ordered tuples is the space which is *the* domain of the function. Similarly, a function or procedure which might naturally be conceived as having multiple outputs is described as having a single output which is an ordered tuple of the “natural” outputs. The “dimension” of the tuple space which is the domain or range of a function indicates the number of individual natural inputs or outputs, respectively. The top level of tupling in a domain or range can be decomposed to obtain the individual input domains or output ranges. (See also 4.2.7.)

The range of a mathematical function is nominal to the extent that any mathematical space which contains all the actual output values may be specified as the value for the range attribute. Since mathematical function in this context means mathematical partial function, the mathematical space supplied as the value for the domain attribute is also technically nominal. However, this is somewhat deceptive. To the greatest extent that is practical, the schema requires a function domain to be the space of all “valid” inputs or some subset thereof. The cases of valid inputs for which the associated algorithm fails should generally be rare.

The domain and range attributes of the abstract supertype **maths_function** are derived attributes. For each subtype, the schema derives appropriate values from the nature of the implied algorithm and explicit attributes of that subtype. For some subtypes, the explicit attributes directly determine the domain and/or range. In any case, the effective domain or range may be further restricted by explicitly composing the function with a restriction function.

The principal effect of the values of domain and range attributes is in determining to which classes of functions the function will belong and, consequently, which roles it might fill in constructing additional functions.

© ISO 2002 — All rights reserved

The actual range space of a given mathematical function on an explicit domain space is the unique mathematical space which contains all and only the output values actually produced by the function from members of that domain. The actual range space is a subspace of every legitimate range space that might be specified for the function. Although mathematically well-defined, the image space is often ill-defined computationally.

The actual domain space of a given mathematical function refers to the unique mathematical space which contains all and only the input values for which the function's algorithm actually produces an output other than the indeterminate value '?'. It, too, is often ill-defined computationally.

4.2.7 Spaces of one-tuples

Mathematicians typically identify a space X and the space X^1 of all one-tuples of elements of X via the obvious canonical natural isomorphism. However, when this is done it becomes impossible to distinguish between the domains of a function $f(x, y)$ which takes two real numbers as input and a function $g((x, y))$ which takes a single input which is a pair of real numbers. These two cases have quite distinct domains for computer programming purposes. Consequently, this schema does maintain the distinction between the domain of f, which is R^2, and the domain of g, which is $(R^2)^1$.

4.2.8 Array function

An array function is a mathematical function whose domain is either a subscript space or the space of one-tuples from a subscript space. A subscript space is a finite Cartesian product of finite intervals of integers. The set of valid subscript tuples for a multi-dimensional array is a subscript space. Given an ordered tuple of subscripts as input, an array function produces a corresponding value. However, an array function need not have a storage location allocated for every possible subscript tuple. Moreover, varying numbers of dimensions can be accommodated in a uniform manner without resorting to recursive constructs such as "ARRAY [1:4] OF ARRAY [0:3] OF ...". Array functions generalize, unify, and support efficient representations of the many specializations of the multi-dimensional array concept.

NOTE Since array functions which are not table functions are uncommon, this part of ISO 10303 primarily provides facilities for representing table functions. Any array function can be represented by reindexing the corresponding table function using the type **reindexed_array_function**. For an example of the representation of a general array function, see example 3 in 4.5.53.

4.2.9 Table function

A table function is an array function in which all the finite intervals of integers appearing in the domain start at zero, or all start at one. In practice, almost all instances of array functions are table functions. Table functions occur when the subscript tuple inputs have no significance beyond establishing relative position. In such cases it is most convenient to use ordinal positions (all subscript ranges start at one) or relative positions (all subscript ranges start at zero). Since both choices are widely used by applications, both are supported.

4.2.10 Matrix

Within this schema, matrix is synonymous with table function defined on a two-dimensional subscript space. Matrices are visualized as rectangular tables of values and much of the terminology is based on this visualization. A position in a matrix is a pair of integers belonging to the subscript domain. The first integer of the pair that constitute a matrix position is the

© ISO 2002 — All rights reserved

row position and the second is the column position. The matrix entry at a position is the value produced as output by the table function when it is applied to the position. Terms for important subsets of positions are also applied to the corresponding subsets of entries. A row is the set of all positions with the same row position. A column is the set of all positions with the same column position. The main diagonal is the set of all positions for which the row position equals the column position. The upper triangle is the set of positions for which the row position is less than or equal to the column position. The strictly upper triangle is the set of positions for which the row position is less than the column position. The lower triangle is the set of positions for which the row position is greater than or equal to the column position. The strictly lower triangle is the set of positions for which the row position is greater than the column position. A diagonal is a set of positions for which the signed difference between the column position and the row position is a given integer constant. This signed difference is the diagonal "position" and is referenced when discussing diagonals "above" or "below" the main diagonal.

NOTE The upper or lower "triangle" could, in the case of a non-square matrix, actually look more like a trapezoid. The word "triangle" will however still be used.

4.2.11 Inputs and Parameters

When a mathematical function is defined by means of an expression, it is sometimes necessary to view the variables as being of two types: function inputs and function parameters. The function parameters are conceived as having been given specific (but, so far, undetermined) values from their corresponding domains, and the function is viewed as having only the remaining inputs as its inputs.

In informal mathematics one might define a function f as follows:

> Let $f(x) = a \cdot x + b$, where a and b are constants.

One might then describe the function f as a general linear function from R^1 to R^1. However, if one examines the defining expression out of context, it appears to describe a quadratic function of three variables, not a linear function of one variable. The expression alone does not provide enough information to capture the mathematical intent. In the informal language of inputs and parameters used above, one would say that x is being viewed as an input while a and b are viewed as parameters.

The formalism which does capture what is occurring here is the Lambda Calculus, which would express the function definition above as:

$$f \equiv \lambda x(a \cdot x + b)$$

This notation uses a logical quantifier, λ, to make clear that the role of the variable x has changed to that of a bound variable. It is now just a placeholder in the expression for the actual input to the function. At the same time, the semantics of the expression has changed and it now denotes a real-valued function of one real variable rather than a real number. This function object is undetermined in that it still depends on the values chosen for the free variables (parameters) a and b. But the function object does not depend on x and it makes no sense to talk about choosing a value for x. Alternatively, one might say that the function object depends on all the possible values for x simultaneously.

The **abstracted_expression_function** subtype explicitly represents the quantifier operation which converts an expression into a function by identifying the variables which are to be treated as function inputs and by specifying their order. Any other variables in the expression are function parameters.

© ISO 2002 — All rights reserved

4.2.12 Function evaluation

A mathematical function is evaluated at inputs to produce the corresponding outputs. The details of the implementation of this process are out of the scope of this standard. However, when such a process is to be evaluated for conformance with this part of ISO 10303, the results shall be equivalent to the following:

First the tuple of inputs is tested to determine whether or not it is in the domain space of the function. If it is not, then the evaluation process ceases and the indeterminate value is produced as output. Second, the functional relationship is invoked to find the output tuple corresponding to the input tuple, if any. If the functional relationship is defined by an algorithm, this means that the algorithm is carried out to produce the corresponding tuple of outputs. If this process fails, the evaluation process ceases and the indeterminate value is produced as output. Third, the tuple of outputs is tested to determine whether or not it is in the range space of the function. If it is not, the indeterminate value is produced as output. Otherwise, the values produced by the second stage are the outputs.

NOTE 1 The price of explicitly providing information about domains and ranges of functions is the need to ensure its validity by incorporating it in the evaluation process.

NOTE 2 Testing a value for membership in a mathematical space may itself require the evaluation of explicit or implicit functions in a recursive manner.

NOTE 3 The general problem of detecting evaluation processes which will not terminate is known to be unsolvable. It is in the self interest of all users to avoid creating or communicating functions whose evaluation does not terminate.

4.2.13 Function application

Function application is the process of forming an expression from a function and an appropriate set of inputs. The term may also be used for an expression resulting from the process. A function application expression denotes the first output from the evaluation of the function on those inputs.

NOTE In most contexts, the term “function” is restricted to algorithms which only have one output, and “first output” is synonymous with “the output”. This part of ISO 10303 uses the term in the wider sense which includes algorithms with multiple outputs, and must therefore adopt this refinement to harmonize the computer programming situation with the more traditional one.

4.3 Mathematical functions schema constant definitions

The following constants are used repeatedly in the remainder of the formal specification of this part of ISO 10303.

NOTE Implementor information: Since schema constants cannot be referenced from ISO 10303-21 files, these constants are only useful within the formal specifications. Instantiations of product data models must reproduce these constant instances as needed. Only one instance of any constant is ever required in a model, although it does no harm if an implementation generates several.

EXPRESS specification:

```
*)
CONSTANT
```


© ISO 2002 — All rights reserved

```
(*
```

4.3.1 schema_prefix

This string constant defines a frequently needed prefix for type names.

EXPRESS specification:

```
*)
  schema_prefix : STRING := 'MATHEMATICAL_FUNCTIONS_SCHEMA.';
(*
```

4.3.2 the elementary spaces

These instances of **elementary_space** represent unique and frequently referenced mathematical spaces.

EXPRESS specification:

```
*)
  the_integers        : elementary_space := make_elementary_space(es_integers);
  the_reals           : elementary_space := make_elementary_space(es_reals);
  the_complex_numbers : elementary_space := make_elementary_space(es_complex_numbers);
  the_numbers         : elementary_space := make_elementary_space(es_numbers);
  the_logicals        : elementary_space := make_elementary_space(es_logicals);
  the_booleans        : elementary_space := make_elementary_space(es_booleans);
  the_strings         : elementary_space := make_elementary_space(es_strings);
  the_binarys         : elementary_space := make_elementary_space(es_binarys);
  the_maths_spaces    : elementary_space := make_elementary_space(es_maths_spaces);
  the_generics        : elementary_space := make_elementary_space(es_generics);
(*
```

4.3.3 the empty space

An instance of **finite_space** representing the unique mathematical space with no elements is frequently referenced.

EXPRESS specification:

```
*)
  the_empty_space : finite_space := make_finite_space([]);
(*
```

4.3.4 real intervals

These instance of **maths_space** represent unique and frequently referenced intervals of real numbers.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
  the_nonnegative_reals         : real_interval_from_min :=
    make_real_interval_from_min(0.0, closed);
  the_zero_one_interval         : finite_real_interval := make_finite_real_interval(
    0.0, closed, 1.0, closed);
  the_zero_pi_interval          : finite_real_interval := make_finite_real_interval(
    0.0, closed, pi, closed);
  the_neg1_one_interval         : finite_real_interval := make_finite_real_interval(
    -1.0, closed, 1.0, closed);
  the_neghalfpi_halfpi_interval : finite_real_interval := make_finite_real_interval(
    -0.5*pi, closed, 0.5*pi, closed);
  the_negpi_pi_interval         : finite_real_interval := make_finite_real_interval(
    -pi, open, pi, closed);
(*
```

4.3.5 tuple spaces

These instances of **tuple_space** represent unique and frequently referenced mathematical tuple spaces.

EXPRESS specification:

```
*)
  the_zero_tuple_space : listed_product_space := make_listed_product_space([]);
  the_tuples           : extended_tuple_space := make_extended_tuple_space(
    the_zero_tuple_space, the_generics);
  the_integer_tuples   : extended_tuple_space := make_extended_tuple_space(
    the_zero_tuple_space, the_integers);
  the_real_tuples      : extended_tuple_space := make_extended_tuple_space(
    the_zero_tuple_space, the_reals);
  the_complex_tuples   : extended_tuple_space := make_extended_tuple_space(
    the_zero_tuple_space, the_complex_numbers);
(*
```

4.3.6 empty values

These special “empty” values of **maths_value** are frequently referenced.

EXPRESS specification:

```
*)
  the_empty_maths_tuple      : maths_tuple := [];
  the_empty_maths_value      : maths_value := the_empty_maths_tuple;
  the_empty_atom_based_tuple : atom_based_tuple := [];
  the_empty_atom_based_value : atom_based_value := the_empty_atom_based_tuple;
(*
```

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
END_CONSTANT;
(*
```

NOTE Since entity instance identity has almost no significance in this schema, the repeated use of the same constant entity instance is acceptable and reduces overhead.

4.4 Mathematical functions schema type definitions

4.4.1 nonnegative_integer

This type provides a named type for the nonnegative integers.

EXPRESS specification:

```
*)
TYPE nonnegative_integer = INTEGER;
WHERE nonnegativity: SELF >= 0;
END_TYPE;
(*
```

4.4.2 positive_integer

This type provides a named type for the positive integers.

EXPRESS specification:

```
*)
TYPE positive_integer = nonnegative_integer;
WHERE positivity: SELF > 0;
END_TYPE;
(*
```

4.4.3 zero_or_one

This subset type of the nonnegative integers is used to indicate whether the derived standard indexing for a table function should start from zero or from one.

EXPRESS specification:

```
*)
TYPE zero_or_one = nonnegative_integer;
WHERE in_range: (SELF = 0) OR (SELF = 1);
END_TYPE;
(*
```

© ISO 2002 — All rights reserved

4.4.4 one_or_two

This subset type of the positive integers is used to indicate whether the summation index for a linear function is one or two.

EXPRESS specification:

```
*)
TYPE one_or_two = positive_integer;
WHERE in_range: (SELF = 1) OR (SELF = 2);
END_TYPE;
(*
```

4.4.5 local names for simple types

These type definitions provide named types for the EXPRESS simple types.

NOTE In order to create the "generic" type **maths_value** (see 4.4.11) and other similar "supertypes" using the EXPRESS select construct, named types must be defined for the EXPRESS simple types.

EXPRESS specification:

```
*)
TYPE maths_number = NUMBER;
END_TYPE;
TYPE maths_real = REAL;
END_TYPE;
TYPE maths_integer = INTEGER;
END_TYPE;
TYPE maths_logical = LOGICAL;
END_TYPE;
TYPE maths_boolean = BOOLEAN;
END_TYPE;
TYPE maths_string = STRING;
END_TYPE;
TYPE maths_binary = BINARY;
END_TYPE;
(*
```

4.4.6 maths_simple_atom

This type encompasses any EXPRESS simple value.

EXPRESS specification:

```
*)
TYPE maths_simple_atom = SELECT
  (maths_number,
   maths_real,
   maths_number,
   maths_logical,
```


© ISO 2002 — All rights reserved

```
   maths_boolean,
   maths_string,
   maths_binary);
END_TYPE;
(*
```

4.4.7 maths_atom

This type encompasses any EXPRESS simple value and any enumeration item of any EXPRESS enumeration type defined in this schema. It is a foundation type for the recursive constructions for types **atom_based_value** and **maths_value**.

EXPRESS specification:

```
*)
TYPE maths_atom = SELECT
  (maths_simple_atom,
   maths_enum_atom);
END_TYPE;
(*
```

4.4.8 atom_based_tuple

This type provides representations for arbitrary finite tuples of **atom_based_value** values.

EXPRESS specification:

```
*)
TYPE atom_based_tuple = LIST OF atom_based_value;
END_TYPE;
(*
```

4.4.9 atom_based_value

This type represents all values constructed from **maths_atom** values by recursive formations of tuples. The EXPRESS representation is arbitrarily nested lists of atoms or the atoms themselves.

EXPRESS specification:

```
*)
TYPE atom_based_value = SELECT
  (maths_atom,
   atom_based_tuple);
END_TYPE;
(*
```

NOTE Since there is no mechanism for self reference, the values are necessarily finite and acyclic. Note, in particular, that no entity instances can participate in values of this type.

© ISO 2002 — All rights reserved

4.4.10 maths_tuple

This type represents the mathematical ordered tuple.

NOTE 1 The semantics of mathematical ordered tuple differs subtly from that of the EXPRESS list aggregate in that there are no notions of creation or destruction or element insertion or removal operations.

NOTE 2 In the context of formal mathematics, there is a single object which can be called the zero-tuple. It is naturally represented by a zero-length list. This object is permitted in the type **maths_tuple**, but not in the other tuple types.

EXPRESS specification:

```
*)
TYPE maths_tuple = LIST [0:?] OF maths_value;
END_TYPE;
(*
```

4.4.11 maths_value

This type is intended to encompass any data value which could be an input to a mathematical function, an output from a mathematical function, or useful in defining a mathematical function.

NOTE The EXPRESS type ”generic“ would be appropriate if it were allowed. Instead, applications needing to use data values not belonging to this type will have to define additional subtypes of the supertypes provided in this schema.

EXPRESS specification:

```
*)
TYPE maths_value = SELECT
  (atom_based_value,
   maths_tuple,
   generic_expression);
WHERE
  constancy: NOT ('GENERIC_EXPRESSION' IN stripped_typeof(SELF)) OR
             expression_is_constant(SELF);
END_TYPE;
(*
```

Formal propositions:

constancy: If a **maths_value** is a **generic_expression** then it shall denote a constant mathematical object.

4.4.12 maths_expression

This type is intended to encompass any expression recognizable as mathematical.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
TYPE maths_expression = SELECT
  (atom_based_value,
   maths_tuple,
   generic_expression);
END_TYPE;
(*
```

4.4.13 maths_function_select

This type enables the use of either **elementary_function_enumerators** values or **maths_function** instances to reference a mathematical function in certain attributes of **parallel_composed_function** and **function_application**.

NOTE This select type enables a more efficient representation of an enormous variety of mathematical functions. In particular, it reduces the number of entity instances required.

EXPRESS specification:

```
*)
TYPE maths_function_select = SELECT
  (maths_function,
   elementary_function_enumerators);
END_TYPE;
(*
```

4.4.14 input_selector

This type of **positive_integer** adds the special semantics of denoting a particular input of a function of several inputs, or a particular component of the first input when there is only one input and that input is a tuple. The positive integer value indicates the ordinal position of the selected input in the list of inputs or in the list of components of the sole tuple input. Values larger than the number of inputs or number of components, respectively, shall be prevented by rules in each context using this type.

NOTE 1 It is normally the case that either all the inputs to a function have been prepackaged as a single input tuple, or the function has multiple inputs, or the single input is not a tuple. If there is a single input that is a tuple and it is desired to select that single input rather than one of its components, then the desired selector function is just an identity function on that domain, and an appropriate instance of **restriction_function** may be used instead.

NOTE 2 This type is used by **selector_function**, **partial_derivative_function**, and **definite_integral_function**.

EXPRESS specification:

```
*)
TYPE input_selector = positive_integer;
END_TYPE;
(*
```

© ISO 2002 — All rights reserved

4.4.15 elementary_space_enumerators

This enumeration type provides unique nominal values to represent the mathematical spaces associated with the EXPRESS simple types, the complex numbers and the generic type.

<u>EXPRESS specification</u>:

```
*)
TYPE elementary_space_enumerators = ENUMERATION OF
  (es_numbers,
   es_complex_numbers,
   es_reals,
   es_integers,
   es_logicals,
   es_booleans,
   es_strings,
   es_binarys,
   es_maths_spaces,
   es_maths_functions,
   es_generics);
END_TYPE;
(*
```

<u>Enumerated item definitions</u>:

es_numbers: Enumeration item used to identify the mathematical space of all numbers.

es_complex_numbers: Enumeration item used to identify the mathematical space of all complex numbers.

es_reals: Enumeration item used to identify the mathematical space of all real numbers.

es_integers: Enumeration item used to identify the mathematical space of all integers.

es_logicals: Enumeration item used to identify the mathematical space of all EXPRESS type `LOGICAL` values.

es_booleans: Enumeration item used to identify the mathematical space of all EXPRESS type `BOOLEAN` values.

es_strings: Enumeration item used to identify the mathematical space of all EXPRESS type `STRING` values.

es_binarys: Enumeration item used to identify the mathematical space of all EXPRESS type `BINARY` values.

es_maths_spaces: Enumeration item used to identify the mathematical space of all mathematical spaces representable by instances of **maths_space**.

es_maths_functions: Enumeration item used to identify the mathematical space of all mathematical functions representable by instances of **maths_function**.

es_generics: Enumeration item used to identify the mathematical space of all mathematical values representable using this schema.

NOTE The "space of all numbers" consists of all mathematical values corresponding to EXPRESS type `NUMBER` values and the complex numbers.

 © ISO 2002 — All rights reserved

4.4.16 ordering_type

This enumeration type indicates which of the two most natural linear "ascending" orderings on the members of a subscript space is to be used. The enumeration identifier **by_rows** indicates a lexicographic order in which the first subscript is most significant and the last subscript varies most rapidly. The enumeration identifier **by_columns** indicates an order in which the last subscript is most significant and the first subscript varies most rapidly.

NOTE The enumeration identifiers were chosen to be meaningful in the most common case, that of subscript pairs for matrices. Traditionally, the first index indicates the row position and the second the column position. Languages which define multi-dimensional arrays by a recursive application of a one-dimensional array construct imply the **by_rows** ordering. The FORTRAN programming language specified the **by_columns** ordering.

EXPRESS specification:

```
*)
TYPE ordering_type = ENUMERATION OF
  (by_rows,
   by_columns);
END_TYPE;
(*
```

Enumerated item definitions:

by_rows: This enumeration value indicates that the subscript tuples are in ascending left-to-right-lexicographic order with the last subscript varying most rapidly.

by_columns: This enumeration value indicates that the subscript tuples are in ascending right-to-left-lexicographic order with the first subscript varying most rapidly.

EXAMPLE Consider the twelve subscript tuples for a 2 by 2 by 3 table indexed from 1. Table 2 shows the two linear orderings of these triples indicated by values **by_rows** and **by_columns**.

Table 2 – Orderings indicated by ordering_type

Ordinal	by_rows	by_columns
1	[1,1,1]	[1,1,1]
2	[1,1,2]	[2,1,1]
3	[1,1,3]	[1,2,1]
4	[1,2,1]	[2,2,1]
5	[1,2,2]	[1,1,2]
6	[1,2,3]	[2,1,2]
7	[2,1,1]	[1,2,2]
8	[2,1,2]	[2,2,2]
9	[2,1,3]	[1,1,3]
10	[2,2,1]	[2,1,3]
11	[2,2,2]	[1,2,3]
12	[2,2,3]	[2,2,3]

© ISO 2002 — All rights reserved

4.4.17 lower_upper

This enumeration type is used to indicate whether the elements below the main diagonal of a matrix are being considered or the elements above. In other words, whether the ordinal positions with the row ordinal greater than or equal to the column ordinal are being considered or the ordinal positions with the row ordinal less than or equal to the column ordinal.

When used in a higher-dimensional context, the enumeration value **lower** indicates the set of ordinal positions for which the ordinal indices are non-increasing, and the enumeration value **upper** indicates the set of ordinal positions for which the ordinal indices are non-decreasing.

NOTE Other context information in a given use of this type may further restrict the set of positions by changing all the inequalities to strict inequalities. Such restrictions are described by using the terms "strictly lower" and "strictly upper".

EXPRESS specification:

```
*)
TYPE lower_upper = ENUMERATION OF
  (lower,
   upper);
END_TYPE;
(*
```

Enumerated item definitions:

lower: The lower triangle of a matrix is to be considered. That is, the elements which have their row ordinal greater than or equal to their column ordinal.

upper: The upper triangle of a matrix is to be considered. That is, the elements which have their row ordinal less than or equal to their column ordinal.

4.4.18 symmetry_type

This enumeration type is used to indicate the kind of symmetry present in a symmetric matrix. Values indicating identity symmetry, skew symmetry, Hermitian symmetry and skew-Hermitian symmetry are defined.

EXPRESS specification:

```
*)
TYPE symmetry_type = ENUMERATION OF
  (identity,
   skew,
   hermitian,
   skew_hermitian);
END_TYPE;
(*
```

 © ISO 2002 — All rights reserved

Enumerated item definitions:

identity: The element in position $[j, k]$ is the same as the element in position $[k, j]$.

skew: The element in position $[j, k]$ is the numeric negative of the element in position $[k, j]$.

hermitian: The element in position $[j, k]$ is the complex conjugate of the element in position $[k, j]$.

skew_hermitian: The element in position $[j, k]$ is the negative complex conjugate of the element in position $[k, j]$.

4.4.19 elementary_function_enumerators

This enumeration type provides unique nominal values identifying a wide collection of elementary mathematical functions and operators. This type is used in constructing expressions and **elementary_function** instances. The enumeration items denoting trigonometric functions and their inverses all assume angle measures in radians.

The naming convention is to use a suffix to distinguish the argument type(s) for those functions or operations which occur in more than one type domain. The suffix "_i" indicates the operation is applied to integers, the suffix "_r" indicates the operation is applied to real numbers, the suffix "_c" indicates the operation is applied to complex numbers, the suffix "_s" indicates the operation is applied to strings, the suffix "_b" indicates the operation is applied to binarys, the suffix "_t" indicates the operation is applied to tuples, the suffix "_it" indicates the operation is applied to integer tuples, the suffix "_rt" indicates the operation is applied to real tuples, and the suffix "_ct" indicates the operation is applied to complex number tuples . In the case of exponentiation, the integer exponent case, which can be interpreted purely in terms of repeated multiplication and division, has been distinguished using suffixes "_i", "_ri" and "_ci", for the cases of integer, real and complex numbers raised to integral powers, with results in the same space as the base.

The six comparison operators are abbreviated in the enumeration item names as follows: "eq" abbreviates "equal" (=), "ne" abbreviates "not equal" (<> or $\neq$), "gt" abbreviates "greater than" (>), "lt" abbreviates "less than" (<), "ge" abbreviates "greater than or equal" (>= or $\geq$), and "le" abbreviates "less than or equal" (<= or $\leq$).

NOTE 1 The only difference between functions and operators is in the notation that is usually used. For the purposes of this schema, the two terms are synonymous.

NOTE 2 Many of these enumeration items correspond to EXPRESS language functions, procedures or operations. Many of the EXPRESS operators are "overloaded", that is, they are unions of several mathematical functions defined on different domains. For the purposes of this schema, the individual operators are identified.

NOTE 3 Consider the following differences in the exponentiation operations: Two to the power minus two, using the function indicated by **ef_exponentiate_i**, is the integer zero obtained from the integer division of one by four. Two to the power minus two, using the function indicated by **ef_exponentiate_ri**, is the real number 0.25 exactly. Two to the power minus two, using the function indicated by **ef_exponentiate_r**, is a real number which is very close to 0.25, but which might not be exactly 0.25 on all systems because of computational round-off error.

NOTE 4 The EXPRESS procedures **insert** and **remove** each have only one VAR parameter, so they correspond directly to functions. The notion that there is some sense in which the input and output aggregates are the same "instance" is not recognized in the mathematical function.

NOTE 5 The EXPRESS aggregate operators - intersection, union, difference, subset, and superset -

© ISO 2002 — All rights reserved

rely in part on the EXPRESS "in" operator, which relies on instance equality rather than value equality. Similarly, the "rolesof" function and the "usedin" function depend explicitly on instance equality, and on an implicit current population of instances. Consequently, they do not correspond to mathematical functions (in the sense used in this schema). The "hibound" and "lobound" functions depend on the type definitions in a schema rather than on the intrinsic structure of a data value, and have also been omitted.

NOTE 6 A useful extension to this part of ISO 10303 would describe a mathematical model of an EXPRESS-based computational environment and an EXPRESS representation thereof. Such an extension would support notions of entity instances and instance identity within a specified schema context and model state and would support the communication of functions defined on model states.

EXPRESS specification:

```
*)
TYPE elementary_function_enumerators = ENUMERATION OF
  (ef_and, ef_or, ef_not, ef_xor,
   ef_negate_i, ef_add_i, ef_subtract_i, ef_multiply_i, ef_divide_i, ef_mod_i,
   ef_exponentiate_i, ef_eq_i, ef_ne_i, ef_gt_i, ef_lt_i, ef_ge_i, ef_le_i,
   ef_abs_i, ef_max_i, ef_min_i, ef_if_i,
   ef_negate_r, ef_reciprocal_r, ef_add_r, ef_subtract_r, ef_multiply_r,
   ef_divide_r, ef_mod_r, ef_exponentiate_r, ef_exponentiate_ri,
   ef_eq_r, ef_ne_r, ef_gt_r, ef_lt_r, ef_ge_r, ef_le_r, ef_abs_r,
   ef_max_r, ef_min_r, ef_acos_r, ef_asin_r, ef_atan2_r, ef_cos_r, ef_exp_r,
   ef_ln_r, ef_log2_r, ef_log10_r, ef_sin_r, ef_sqrt_r, ef_tan_r, ef_if_r,
   ef_form_c, ef_rpart_c, ef_ipart_c,
   ef_negate_c, ef_reciprocal_c, ef_add_c, ef_subtract_c, ef_multiply_c,
   ef_divide_c, ef_exponentiate_c, ef_exponentiate_ci, ef_eq_c, ef_ne_c,
   ef_conjugate_c, ef_abs_c, ef_arg_c, ef_cos_c, ef_exp_c, ef_ln_c, ef_sin_c,
   ef_sqrt_c, ef_tan_c, ef_if_c,
   ef_subscript_s, ef_eq_s, ef_ne_s, ef_gt_s, ef_lt_s, ef_ge_s, ef_le_s,
   ef_subsequence_s, ef_concat_s, ef_size_s, ef_format, ef_value, ef_like, ef_if_s,
   ef_subscript_b, ef_eq_b, ef_ne_b, ef_gt_b, ef_lt_b, ef_ge_b, ef_le_b,
   ef_subsequence_b, ef_concat_b, ef_size_b, ef_if_b,
   ef_subscript_t, ef_eq_t, ef_ne_t, ef_concat_t, ef_size_t,
   ef_entuple, ef_detuple, ef_insert, ef_remove, ef_if_t,
   ef_sum_it, ef_product_it,
   ef_add_it, ef_subtract_it, ef_scalar_mult_it, ef_dot_prod_it,
   ef_sum_rt, ef_product_rt,
   ef_add_rt, ef_subtract_rt, ef_scalar_mult_rt, ef_dot_prod_rt, ef_norm_rt,
   ef_sum_ct, ef_product_ct,
   ef_add_ct, ef_subtract_ct, ef_scalar_mult_ct, ef_dot_prod_ct, ef_norm_ct,
   ef_if, ef_ensemble, ef_member_of);
END_TYPE;
(*
```

Enumerated item definitions:

ef_and: Enumeration member denoting the EXPRESS logical and operation.

ef_or: Enumeration member denoting the EXPRESS logical or operation.

ef_not: Enumeration member denoting the EXPRESS logical not operation.

ef_xor: Enumeration member denoting the EXPRESS logical xor operation.

ef_negate_i: Enumeration member denoting the integer negation operation.

 © ISO 2002 — All rights reserved

ef_add_i: Enumeration member denoting the integer addition operation.

ef_subtract_i: Enumeration member denoting the integer subtraction operation.

ef_multiply_i: Enumeration member denoting the integer multiplication operation.

ef_divide_i: Enumeration member denoting the integer division operation. This is the same as the EXPRESS “div” operation.

ef_mod_i: Enumeration member denoting the integer modulo operation. This is the same as the EXPRESS “mod” operation.

ef_exponentiate_i: Enumeration member denoting the integer exponentiation operation implemented using integer multiplication and division.

ef_eq_i: Enumeration member denoting the integer equality relation.

ef_ne_i: Enumeration member denoting the integer inequality relation.

ef_gt_i: Enumeration member denoting the integer greater than relation.

ef_lt_i: Enumeration member denoting the integer less than relation.

ef_ge_i: Enumeration member denoting the integer greater than or equal relation.

ef_le_i: Enumeration member denoting the integer less than or equal relation.

ef_abs_i: Enumeration member denoting the integer absolute value function.

ef_max_i: Enumeration member denoting the integer maximum value function.

ef_min_i: Enumeration member denoting the integer minimum value function.

ef_if_i: Enumeration member denoting the integer ternary conditional operation (as in the C programming language).

ef_negate_r: Enumeration member denoting the real negation operation.

ef_reciprocal_r: Enumeration member denoting the real reciprocal operation.

ef_add_r: Enumeration member denoting the real addition operation.

ef_subtract_r: Enumeration member denoting the real subtraction operation.

ef_multiply_r: Enumeration member denoting the real multiplication operation.

ef_divide_r: Enumeration member denoting the real division operation.

ef_mod_r: Enumeration member denoting the real modulo operation. For real numbers a and $b \neq 0$, $a \textbf{ mod } b \equiv a - \lfloor a/b \rfloor * b$, where $\lfloor x \rfloor$ is the greatest integer less than or equal to x.

ef_exponentiate_r: Enumeration member denoting the real exponentiation operation.

ef_exponentiate_ri: Enumeration member denoting exponentiation operation restricted to reals raised to integer powers.

ef_eq_r: Enumeration member denoting the real equality relation.

ef_ne_r: Enumeration member denoting the real inequality relation.

ef_gt_r: Enumeration member denoting the real greater than relation.

ef_lt_r: Enumeration member denoting the real less than relation.

ef_ge_r: Enumeration member denoting the real greater than or equal relation.

ef_le_r: Enumeration member denoting the real less than or equal relation.

© ISO 2002 — All rights reserved

ef_abs_r: Enumeration member denoting the real absolute value function.

ef_max_r: Enumeration member denoting the real maximum value function.

ef_min_r: Enumeration member denoting the real minimum value function.

ef_acos_r: Enumeration member denoting the real inverse cosine function.

ef_asin_r: Enumeration member denoting the real inverse sine function.

ef_atan2_r: Enumeration member denoting the real four-quadrant inverse tangent function. Its two arguments y and x, which shall not both be zero, determine a point (x, y) in the real plane. The signed angle θ between the positive x-axis and the ray from the origin through that point, measured in radians and lying in the range $-\pi < \theta \leq \pi$ shall be returned. When the first input, y, is positive, the output is positive.

ef_cos_r: Enumeration member denoting the real cosine function.

ef_exp_r: Enumeration member denoting the real exponential function.

ef_ln_r: Enumeration member denoting the real natural logarithm function.

ef_log2_r: Enumeration member denoting the real logarithm base two function.

ef_log10_r: Enumeration member denoting the real logarithm base ten function.

ef_sin_r: Enumeration member denoting the real sine function.

ef_sqrt_r: Enumeration member denoting the real square root function.

ef_tan_r: Enumeration member denoting the real tangent function.

ef_if_r: Enumeration member denoting the real ternary conditional operation (as in the C programming language).

ef_form_c: Enumeration member denoting the function which forms a complex number from two reals representing the real and imaginary parts.

ef_rpart_c: Enumeration member denoting the function which returns the real part of a complex number.

ef_ipart_c: Enumeration member denoting the function which returns the imaginary part of a complex number.

ef_negate_c: Enumeration member denoting the complex negation operation.

ef_reciprocal_c: Enumeration member denoting the complex reciprocal operation.

ef_add_c: Enumeration member denoting the complex addition operation.

ef_subtract_c: Enumeration member denoting the complex subtraction operation.

ef_multiply_c: Enumeration member denoting the complex multiplication operation.

ef_divide_c: Enumeration member denoting the complex division operation.

ef_exponentiate_c: Enumeration member denoting the complex exponentiation operation.

ef_exponentiate_ci: Enumeration member denoting exponentiation operation restricted to complexs raised to integer powers.

ef_eq_c: Enumeration member denoting the complex equality relation.

ef_ne_c: Enumeration member denoting the complex inequality relation.

ef_conjugate_c: Enumeration member denoting the complex conjugate operation.

 © ISO 2002 — All rights reserved

ef_abs_c: Enumeration member denoting the complex absolute value function.

ef_arg_c: Enumeration member denoting the complex argument function, which shall be defined by $arg(x + yi) \equiv atan2(y, x)$.

ef_cos_c: Enumeration member denoting the complex cosine function.

ef_exp_c: Enumeration member denoting the complex exponential function.

ef_ln_c: Enumeration member denoting the complex natural logarithm function.

ef_sin_c: Enumeration member denoting the complex sine function.

ef_sqrt_c: Enumeration member denoting the complex square root function producing the square root with positive real part, or, if the real parts are zero, the root with nonnegative imaginary part.

ef_tan_c: Enumeration member denoting the complex tangent function.

ef_if_c: Enumeration member denoting the complex ternary conditional operation (analogous to the corresponding real operation).

ef_subscript_s: Enumeration member denoting the EXPRESS string indexing operation with one index.

ef_eq_s: Enumeration member denoting the string equality relation.

ef_ne_s: Enumeration member denoting the string inequality relation.

ef_gt_s: Enumeration member denoting the string greater than relation.

ef_lt_s: Enumeration member denoting the string less than relation.

ef_ge_s: Enumeration member denoting the string greater than or equal relation.

ef_le_s: Enumeration member denoting the string less than or equal relation.

ef_subsequence_s: Enumeration member denoting the EXPRESS string indexing operation with two indices.

ef_concat_s: Enumeration member denoting the string concatenation operation.

ef_size_s: Enumeration member denoting the string size operation.

ef_format: Enumeration member denoting the EXPRESS formatting function.

ef_value: Enumeration member denoting the EXPRESS string to number conversion function.

ef_like: Enumeration member denoting the EXPRESS string pattern matching operation.

ef_if_s: Enumeration member denoting the string ternary conditional operation (analogous to the corresponding real operation).

ef_subscript_b: Enumeration member denoting the EXPRESS binary indexing operation with one index.

ef_eq_b: Enumeration member denoting the binary equality relation.

ef_ne_b: Enumeration member denoting the binary inequality relation.

ef_gt_b: Enumeration member denoting the binary greater than relation.

ef_lt_b: Enumeration member denoting the binary less than relation.

ef_ge_b: Enumeration member denoting the binary greater than or equal relation.

ef_le_b: Enumeration member denoting the binary less than or equal relation.

© ISO 2002 — All rights reserved

ef_subsequence_b: Enumeration member denoting the EXPRESS binary indexing operation with two indices.

ef_concat_b: Enumeration member denoting the binary concatenation operation.

ef_size_b: Enumeration member denoting the binary size operation.

ef_if_b: Enumeration member denoting the binary ternary conditional operation (analogous to the corresponding real operation).

ef_subscript_t: Enumeration member denoting the tuple indexing operation.

ef_eq_t: Enumeration member denoting the tuple equality relation.

ef_ne_t: Enumeration member denoting the tuple inequality relation.

ef_concat_t: Enumeration member denoting the tuple concatenation operation.

ef_size_t: Enumeration member denoting the tuple size operation.

ef_entuple: Enumeration member denoting the entuple function, that is, the function which collects its input arguments into an ordered tuple and delivers that tuple as its single output.

ef_detuple: Enumeration member denoting the operation returning the all the members of an ordered tuple as explicit independent outputs.

ef_insert: Enumeration member denoting the function version of the EXPRESS insert in list procedure.

ef_remove: Enumeration member denoting the function version of the EXPRESS remove from list procedure.

ef_if_t: Enumeration member denoting the tuple ternary conditional operation (analogous to the corresponding real operation).

ef_sum_it: Enumeration member denoting the operation which sums the members of an integer tuple to produce an integer result.

ef_product_it: Enumeration member denoting the operation which multiplies the members of an integer tuple to produce an integer result.

ef_add_it: Enumeration member denoting the integer tuple addition operation.

ef_subtract_it: Enumeration member denoting the integer tuple subtraction operation.

ef_scalar_mult_it: Enumeration member denoting the integer tuple scalar multiplication operation.

ef_dot_prod_it: Enumeration member denoting the integer tuple dot product operation.

ef_sum_rt: Enumeration member denoting the operation which sums the members of a real tuple to produce a real result.

ef_product_rt: Enumeration member denoting the operation which multiplies the members of a real tuple to produce a real result.

ef_add_rt: Enumeration member denoting the real tuple addition operation.

ef_subtract_rt: Enumeration member denoting the real tuple subtraction operation.

ef_scalar_mult_rt: Enumeration member denoting the real tuple scalar multiplication operation.

ef_dot_prod_rt: Enumeration member denoting the real tuple dot product operation.

ef_norm_rt: Enumeration member denoting the real tuple norm operation.

 © ISO 2002 — All rights reserved

ef_sum_ct: Enumeration member denoting the operation which sums the members of a complex tuple to produce a complex number result.

ef_product_ct: Enumeration member denoting the operation which multiplies the members of a complex tuple to produce a complex number result.

ef_add_ct: Enumeration member denoting the complex tuple addition operation.

ef_subtract_ct: Enumeration member denoting the complex tuple subtraction operation.

ef_scalar_mult_ct: Enumeration member denoting the complex tuple scalar multiplication operation.

ef_dot_prod_ct: Enumeration member denoting the complex tuple dot product operation.

ef_norm_ct: Enumeration member denoting the complex tuple norm operation.

ef_if: Enumeration member denoting the generic ternary conditional operation (analogous to the corresponding real operation).

ef_ensemble: Enumeration member denoting the operation which forms a **finite_space** from its operands.

ef_member_of: Enumeration member denoting the operation which tests a mathematical value for membership in a mathematical space.

4.4.20 open_closed

This enumeration type is used to indicate whether real intervals are topologically open or closed at their endpoints. If the real interval is closed at an endpoint, the endpoint is a member of the real interval; otherwise, the endpoint is not a member of the real interval.

EXPRESS specification:

```
*)
TYPE open_closed = ENUMERATION OF
  (open,
   closed);
END_TYPE;
(*
```

Enumerated item definitions:

open: Indicator that the real interval is topologically open at the associated endpoint, that is, that the boundary point is not a member of the interval.

closed: Indicator that the real interval is topologically closed at the associated endpoint, that is, that the boundary point is a member of the interval.

4.4.21 space_constraint_type

This enumeration type is used to indicate which of three possible constraints on a mathematical space is to be used in a given context. The possibilities are that one space is equal to another, a subspace of another, or a member of another.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
TYPE space_constraint_type = ENUMERATION OF
  (sc_equal,
   sc_subspace,
   sc_member);
END_TYPE;
(*
```

Enumerated item definitions:

sc_equal: Indicator that the subject space shall be equal to the given argument space.

sc_subspace: Indicator that the subject space shall be a subspace, but not necessarily a proper subspace, of the given argument space.

sc_member: Indicator that the subject space shall be member of the given argument space.

4.4.22 repackage_options

This enumeration type is used by **repackaging_function** instances to identify repackaging options for inputs and outputs.

EXPRESS specification:

```
*)
TYPE repackage_options = ENUMERATION OF
  (ro_nochange,
   ro_wrap_as_tuple,
   ro_unwrap_tuple);
END_TYPE;
(*
```

Enumerated item definitions:

ro_nochange: Indicator that no repackaging is to occur.

ro_wrap_as_tuple: Indicator that multiple values are to be repackaged as a single tuple of values.

ro_unwrap_tuple: Indicator that a single tuple value is to be repackaged as multiple values.

4.4.23 extension_options

This enumeration type identifies certain common mathematical mechanisms for extending the exact domain of a function in an unambiguous way to boundary and other technically singular inputs.

NOTE This type is used by **partial_derivative_function** instances to identify which of certain common extensions to the function determined by the strictest notion of differentiability are to be made.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
TYPE extension_options = ENUMERATION OF
  (eo_none,
   eo_cont,
   eo_cont_right,
   eo_cont_left);
END_TYPE;
(*
```

Enumerated item definitions:

eo_none: Indicator that no extension shall occur.

eo_cont: Indicator that extension by continuity shall occur. Wherever an unambiguous value can be computed which preserves continuity, the function shall be defined there and output that value.

eo_cont_right: Indicator that extension by continuity shall occur and, when and where continuity to the right is meaningful, extension by continuity to the right shall occur.

eo_cont_left: Indicator that extension by continuity shall occur and, when and where continuity to the left is meaningful, extension by continuity to the left shall occur.

NOTE Extension by continuity to the right or left is only meaningful in the context of the real numbers. When extension by continuity occurs, the value produced is necessarily the same as the values produced by continuity to the right or continuity to the left.

4.4.24 maths_enum_atom

This select type collects the enumeration types defined in this schema which contain mathematical values and which are not otherwise included in the type used to represent mathematical values.

EXPRESS specification:

```
*)
TYPE maths_enum_atom = SELECT
  (elementary_space_enumerators,
   ordering_type,
   lower_upper,
   symmetry_type,
   elementary_function_enumerators,
   open_closed,
   space_constraint_type,
   repackage_options,
   extension_options);
END_TYPE;
(*
```

© ISO 2002 — All rights reserved

4.4.25 dotted_express_identifier

This type of STRING includes the strings which could syntactically be qualified type names or attribute names in EXPRESS. That is, it includes strings which are composed of one or more EXPRESS identifiers separated by period characters.

EXPRESS specification:

```
*)
TYPE dotted_express_identifier = STRING;
WHERE syntax: dotted_identifiers_syntax(SELF);
END_TYPE;
(*
```

Formal propositions:

syntax: The string shall have the form of a sequence of EXPRESS identifiers separated by full stops.

4.4.26 express_identifier

This type of **dotted_express_identifier** includes only the strings which could syntactically be EXPRESS identifiers. In particular, a value of this type shall begin with a letter and contain only letters, digits and low lines.

EXPRESS specification:

```
*)
TYPE express_identifier = dotted_express_identifier;
WHERE syntax: dot_count(SELF) = 0;
END_TYPE;
(*
```

Formal propositions:

syntax: The string shall begin with a letter and contain only letters, digits, and low lines.

NOTE In order that **express_identifier** be a subtype of **dotted_express_identifier**, the rule is actually defined by requiring the string to be an instance of **dotted_express_identifier** which contains no full stops.

4.4.27 product_space

A **product_space** is a selection from among the entity types representing finite Cartesian product spaces.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
TYPE product_space = SELECT
  (uniform_product_space,
   listed_product_space);
END_TYPE;
(*
```

4.4.28 tuple_space

A **tuple_space** is a selection from among the types representing mathematical spaces whose elements are ordered tuples. This values of this type are those which could occur as domains or ranges of mathematical functions.

NOTE Specifically, the values of this type represent finite Cartesian products of mathematical spaces, and certain infinite unions of such Cartesian products. The latter case arises for functions with an indefinite number of input or output arguments.

EXPRESS specification:

```
*)
TYPE tuple_space = SELECT
  (product_space,
   extended_tuple_space);
END_TYPE;
(*
```

4.4.29 maths_space_or_function

A **maths_space_or_function** is a selection of either a **maths_space** or a **maths_function**.

NOTE This type describes the two choices for the source of the information for determining the domain of a **constant_function**, **selector_function**, or **parallel_composed_function**.

EXPRESS specification:

```
*)
TYPE maths_space_or_function = SELECT
  (maths_space,
   maths_function);
END_TYPE;
(*
```

4.4.30 real_interval

A **real_interval** is a selection from among the entity types representing non-empty, non-trivial, real intervals.

NOTE The empty interval and the single point interval are not representable by this type. All other real intervals have a value-unique representation in this type.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
TYPE real_interval = SELECT
  (real_interval_from_min,
   real_interval_to_max,
   finite_real_interval,
   elementary_space);
WHERE
  WR1: NOT ('ELEMENTARY_SPACE' IN stripped_typeof(SELF)) OR
    (SELF\elementary_space.space_id = es_reals);
END_TYPE;
(*
```

Formal propositions:

WR1: If the select value is an **elementary_space**, it is the one representing the space of all real numbers.

4.5 Mathematical functions schema entity definitions

4.5.1 quantifier_expression

This abstract type of **multiple_arity_generic_expression** distinguishes those expressions which are quantifier expressions. Certain of the operands of a quantifier expression are restricted to variables. These variables are said to be bound by the quantifier and their semantics is determined by the quantifier. The variables which are bound by a quantifier are not available for substitution operations, that is, they may not be subsequently tied to particular constant values. The semantics of the quantifier expression depends on all the possible values of each of its bound variables rather than varying with the association of variables to particular values.

NOTE 1 This type together with entity type **bound_variable_semantics**, function **free_variables_of** and **expression_is_constant** generically extend the **ISO13584_generic_expressions_schema** to handle operations which involve logical quantifiers.

NOTE 2 Function **has_values_space** identifies which generic expressions pertain to the **mathematical_functions_schema** and, consequently, have associated spaces of possible values given by function **values_space_of**.

EXPRESS specification:

```
*)
ENTITY quantifier_expression
  ABSTRACT SUPERTYPE
  SUBTYPE OF (multiple_arity_generic_expression);
  variables : LIST [1:?] OF UNIQUE generic_variable;
WHERE
  WR1: SIZEOF (QUERY (vrbl <* variables | NOT (vrbl IN
       SELF\multiple_arity_generic_expression.operands))) = 0;
  WR2: SIZEOF (QUERY (vrbl <* variables | NOT ((schema_prefix +
       'BOUND_VARIABLE_SEMANTICS') IN TYPEOF (vrbl.interpretation.semantics)))) = 0;
END_ENTITY;
(*
```

 © ISO 2002 — All rights reserved

Attribute definitions:

variables: The list of variables bound by the quantifier operation.

Formal propositions:

WR1: The variables listed in attribute **variables** are among the operands of the operation.

WR2: The variables listed in attribute **variables** have the semantics of bound variables.

4.5.2 dependent_variable_definition

This subtype of **unary_generic_expression** defines a dependent variable. From the viewpoint of the formal language of mathematical expressions, a dependent variable is merely an abbreviation for a subexpression which is being given special attention because of its significance or frequency of occurence in other expressions.

NOTE Observe that a dependent variable is not a variable at all! In particular, it is not a logical variable, may not be quantified, may not be assigned a value (other than that resulting from the defining expression), and will not appear in lists of used or free variables. It has significance for rendering operations, which should use the associated name instead of rendering the defining expression when a reference to the **dependent_variable_definition** instance is used in another expression, and it has significance for computational operations, which may improve efficiency of computation by saving and reusing its value under appropriate circumstances, but it has no effect on the meaning of the defining expression as a formal expression.

EXPRESS specification:

```
*)
ENTITY dependent_variable_definition
  SUBTYPE OF (unary_generic_expression);
  name        : label;
  description : text;
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The expression defining the dependent variable.

name: The label identifying the dependent variable.

description: Descriptive information for the dependent variable or defining expression.

NOTE The description may also include information about the preferred rendering of the dependent variable or the defining expression in some presentation system such as TeX or XML. Of course, only certain receiving systems would be able to make use of such information.

4.5.3 bound_variable_semantics

This type of **variable_semantics** indicates that a variable is a bound variable whose semantics is determined by the quantifier expression which references the variable in its **variables** attribute.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
ENTITY bound_variable_semantics
  SUBTYPE OF (variable_semantics);
END_ENTITY;
(*
```

Informal propositions:

IP1: Only those variables which are listed in the **variables** attribute of a quantifier expression may have this semantics.

4.5.4 free_variable_semantics

This type of **variable_semantics** indicates that a variable has no further semantics beyond what is implied by its use in an expression. This could occur when the expression itself is the object of interest.

EXPRESS specification:

```
*)
ENTITY free_variable_semantics
  SUBTYPE OF (variable_semantics);
END_ENTITY;
(*
```

4.5.5 complex_number_literal

This type of **generic_literal** represents mathematical complex number constants in expressions.

EXPRESS specification:

```
*)
ENTITY complex_number_literal
  SUBTYPE OF (generic_literal);
  real_part : REAL;
  imag_part : REAL;
END_ENTITY;
(*
```

Attribute definitions:

real_part: The value of the real part of the complex number which is to be represented by this literal in expressions.

imag_part: The value of the imaginary part of the complex number which is to be represented by this literal in expressions.

 © ISO 2002 — All rights reserved

4.5.6 logical_literal

This type of **generic_literal** represents EXPRESS type `LOGICAL` constants in expressions.

EXPRESS specification:

```
*)
ENTITY logical_literal
  SUBTYPE OF (generic_literal);
  lit_value : LOGICAL;
END_ENTITY;
(*
```

Attribute definitions:

lit_value: The EXPRESS value representing the mathematical value which is to be represented by this literal in expressions.

4.5.7 binary_literal

This type of **generic_literal** represents EXPRESS type `BINARY` constants in expressions.

EXPRESS specification:

```
*)
ENTITY binary_literal
  SUBTYPE OF (generic_literal);
  lit_value : BINARY;
END_ENTITY;
(*
```

Attribute definitions:

lit_value: The EXPRESS value representing the mathematical value which is to be represented by this literal in expressions.

4.5.8 maths_enum_literal

This type of **generic_literal** reresents **mathematical_functions_schema** enumeration item constants in expressions.

EXPRESS specification:

```
*)
ENTITY maths_enum_literal
  SUBTYPE OF (generic_literal);
  lit_value : maths_enum_atom;
END_ENTITY;
```

© ISO 2002 — All rights reserved

```
(*
```

Attribute definitions:

lit_value: The EXPRESS value representing the mathematical value which is to be represented by this literal in expressions.

4.5.9 real_tuple_literal

This type of **generic_literal** represents tuples of real constants in expressions.

EXPRESS specification:

```
*)
ENTITY real_tuple_literal
  SUBTYPE OF (generic_literal);
  lit_value : LIST [1:?] OF REAL;
END_ENTITY;
(*
```

Attribute definitions:

lit_value: The EXPRESS value representing the mathematical value which is to be represented by this literal in expressions.

4.5.10 integer_tuple_literal

This type of **generic_literal** represents tuples of integer constants in expressions.

EXPRESS specification:

```
*)
ENTITY integer_tuple_literal
  SUBTYPE OF (generic_literal);
  lit_value : LIST [1:?] OF INTEGER;
END_ENTITY;
(*
```

Attribute definitions:

lit_value: The EXPRESS value representing the mathematical value which is to be represented by this literal in expressions.

4.5.11 atom_based_literal

This type of **generic_literal** represents recursively constructed tuples of atomic constants in expressions. The atomic constants in this schema are all the values of EXPRESS sim-

 © ISO 2002 — All rights reserved

ple types and all the enumeration items of the EXPRESS enumeration types defined in this schema.

NOTE All the constants represented by the preceding subtypes of **generic_literal** may also be represented using **atom_based_literal**. It is recommended that the simpler subtypes be used where possible for the sake of readability. It may also be the case that some applications will not need the general case and not permit its use. Another complication is that an **atom_based_value** with a real value will not be recognized as a **real_numeric_literal** and not be usable in real numeric expressions from **ISO13584_expressions_schema**. The analogous complication exists for single integer, boolean and string values.

EXPRESS specification:

```
*)
ENTITY atom_based_literal
  SUBTYPE OF (generic_literal);
  lit_value : atom_based_value;
END_ENTITY;
(*
```

Attribute definitions:

lit_value: The EXPRESS value representing the mathematical value which is to be represented by this literal in expressions.

4.5.12 maths_tuple_literal

This type of **generic_literal** represents ordered tuples of arbitrary mathematical objects in expressions.

EXPRESS specification:

```
*)
ENTITY maths_tuple_literal
  SUBTYPE OF (generic_literal);
  lit_value : LIST OF maths_value;
END_ENTITY;
(*
```

Attribute definitions:

lit_value: The list of EXPRESS values representing the components of the mathematical ordered tuple which is to be represented by this literal in expressions.

4.5.13 maths_variable

This type of **generic_variable** represents a variable ranging over a given mathematical space of values.

NOTE The identity of a **maths_variable** is derived from its identity as an entity instance, so the

© ISO 2002 — All rights reserved

presence or absence of a name is irrelevant to its role in any expression. However, for many applications, a name contains very useful information, especially in rendering the expression for human readers.

EXPRESS specification:

```
*)
ENTITY maths_variable
  SUBTYPE OF (generic_variable);
  values_space : maths_space;
  name         : label;
WHERE
  WR1: expression_is_constant(values_space);
END_ENTITY;
(*
```

Attribute definitions:

values_space: The mathematical space of possible values of the variable.

Formal propositions:

WR1: The value of attribute **values_space** shall be a constant expression.

4.5.14 maths_real_variable

This type of **maths_variable** and **real_numeric_variable** makes a convenient representation for variables ranging over the real numbers.

EXPRESS specification:

```
*)
ENTITY maths_real_variable
  SUBTYPE OF (maths_variable, real_numeric_variable);
WHERE
  WR1: subspace_of_es(SELF\maths_variable.values_space,es_reals);
END_ENTITY;
(*
```

Formal propositions:

WR1: The value of inherited attribute **values_space** is a subspace of the real numbers.

4.5.15 maths_integer_variable

This type of **maths_variable** and **int_numeric_variable** makes a convenient representation for variables ranging over the integers.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
ENTITY maths_integer_variable
  SUBTYPE OF (maths_variable, int_numeric_variable);
WHERE
  WR1: subspace_of_es(SELF\maths_variable.values_space,es_integers);
END_ENTITY;
(*
```

Formal propositions:

WR1: The value of inherited attribute **values_space** is a subspace of the integers.

4.5.16 maths_boolean_variable

This type of **maths_variable** and **boolean_variable** makes a convenient representation for variables ranging over the booleans.

EXPRESS specification:

```
*)
ENTITY maths_boolean_variable
  SUBTYPE OF (maths_variable, boolean_variable);
WHERE
  WR1: subspace_of_es(SELF\maths_variable.values_space,es_booleans);
END_ENTITY;
(*
```

Formal propositions:

WR1: The value of inherited attribute **values_space** is a subspace of the booleans.

4.5.17 maths_string_variable

This type of **maths_variable** and **string_variable** makes a convenient representation for variables ranging over strings.

EXPRESS specification:

```
*)
ENTITY maths_string_variable
  SUBTYPE OF (maths_variable, string_variable);
WHERE
  WR1: subspace_of_es(SELF\maths_variable.values_space,es_strings);
END_ENTITY;
(*
```

© ISO 2002 — All rights reserved

Formal propositions:

WR1: The value of inherited attribute **values_space** is a subspace of the strings.

4.5.18 function_application

This type of **multiple_arity_generic_expression** represents a mathematical expression that specifies the application of a mathematical function to an appropriate set of arguments. Such a mathematical expression shall denote the first output of evaluation of the function on those arguments.

NOTE In contrast to common usage, this schema admits the possibility of mathematical functions with multiple outputs. Such functions are often called "procedures". If an expression created by applying such a function to arguments is to be interpreted as having a single denotation, then it is hereby declared to denote the first of its outputs.

EXPRESS specification:

```
*)
ENTITY function_application
  SUBTYPE OF (multiple_arity_generic_expression);
  func      : maths_function_select;
  arguments : LIST [1:?] OF maths_expression;
DERIVE
  SELF\multiple_arity_generic_expression.operands : LIST [2:?] OF generic_expression
    := [convert_to_maths_function(func)] + convert_to_operands(arguments);
WHERE
  WR1: function_applicability(func, arguments);
END_ENTITY;
(*
```

Attribute definitions:

func: The function to be applied.

arguments: The list of arguments to which the function is to be applied.

SELF\multiple_arity_generic_expression.operands: The inherited attribute **operands** is derived by prepending the function to the list of arguments.

NOTE For the convenience of users, most of the expression "operations" in this schema use attributes which are not themselves of type **generic_expression** but are of simpler non-entity types. It is easy to derive the corresponding **generic_literal** instances from these types. From these instances, it is easy to derive the inherited **operands** attribute.

Formal propositions:

WR1: The number of arguments shall match the dimension of the function domain and the value spaces of the individual arguments shall be compatible with the corresponding function domain factor spaces.

EXAMPLE To represent the mathematical expression $atan2(y, 1)$, the following instance of **function_application** could be constructed:

 © ISO 2002 — All rights reserved

```
LOCAL
  vrbl_y    : maths_real_variable := make_maths_real_variable(the_reals, 'y');
  atan2_y_1 : function_application := make_function_application(ef_atan2_r,
    [vrbl_y, 1.0]);
END_LOCAL;
```

Two entity instances are constructed. The first creates an instance of **generic_variable** representing a variable ranging over the real numbers and labeled 'y'. The second creates an instance of **generic_expression** representing the application of real-valued function $atan2$ to the list of two arguments consisting of the variable y and the constant 1.0.

4.5.19 maths_space

This abstract supertype includes all the representations for mathematical spaces considered in this schema.

EXPRESS specification:

```
*)
ENTITY maths_space
  ABSTRACT SUPERTYPE OF (ONEOF (elementary_space,
                               finite_integer_interval,
                               integer_interval_from_min,
                               integer_interval_to_max,
                               finite_real_interval,
                               real_interval_from_min,
                               real_interval_to_max,
                               cartesian_complex_number_region,
                               polar_complex_number_region,
                               finite_space,
                               uniform_product_space,
                               listed_product_space,
                               extended_tuple_space,
                               function_space))
  SUBTYPE OF (generic_expression);
END_ENTITY;
(*
```

NOTE Most of the entity declarations in the remainder of this clause will have dual inheritance. They will inherit from either **maths_space** or **maths_function** to acquire their mathematical semantics, and they will inherit from **generic_literal**, **unary_generic_expression**, or **multiple_arity_generic_expression** to identify their syntactical role in a mathematical expression. The choice between modelling something as a complicated constant (inheriting from **generic_literal**) or as a general expression (inheriting from **unary_** or **multiple_arity_generic_expression** has subtle and extensive consequences.

For the applications for which this part of ISO 10303 has been created, there is a strong preference for the semantic simplicity of modeling complicated constants over the semantic complexity of general expressions with many free variables and larger numbers of complexly related instances. In particular, all the entity types representing mathematical spaces are designed to represent specific constant spaces rather than the indeterminate spaces which would result from the presence of free variables in a mathematical space expression. Operations on mathematical spaces may be added to this schema if and whan an application is identified which requires such generality.

© ISO 2002 — All rights reserved

4.5.20 elementary_space

This type of **maths_space** and **generic_literal** is used to represent the elementary mathematical spaces identified by the enumeration type **elementary_maths_space**.

EXPRESS specification:

```
*)
ENTITY elementary_space
  SUBTYPE OF (maths_space, generic_literal);
  space_id : elementary_space_enumerators;
END_ENTITY;
(*
```

Attribute definitions:

space_id: The enumeration item which identifies the space being represented.

4.5.21 finite_integer_interval

This type of **maths_space** and **generic_literal** represents mathematical spaces which are non-empty, finite intervals of integers.

EXPRESS specification:

```
*)
ENTITY finite_integer_interval
  SUBTYPE OF (maths_space, generic_literal);
  min  : INTEGER;
  max  : INTEGER;
DERIVE
  size : positive_integer := max - min + 1;
WHERE
  WR1: min <= max;
END_ENTITY;
(*
```

Attribute definitions:

min: The least integer in the interval.

max: The largest integer in the interval.

size: The derived number of integers which are members of the interval.

Formal propositions:

WR1: The lower bound shall be less than or equal to the upper bound.

NOTE That **generic_literal** is a supertype of **finite_integer_interval** may seem strange at first.

 © ISO 2002 — All rights reserved

The reason is as follows: An instance of **finite_integer_interval** denotes a unique mathematical space. Therefore, it may play the role of a constant in the syntax of mathematical expressions. It is not a "literal" in the ordinary sense, that is, a language token which directly encodes a value, but **generic_literal** is the only choice of supertype from **generic_expressions_schema** which gives these instances their proper role as constants in expressions.

4.5.22 integer_interval_from_min

This type of **maths_space** and **generic_literal** represents semi-infinite integer intervals which contain all integers greater than or equal to a given integer.

EXPRESS specification:

```
*)
ENTITY integer_interval_from_min
  SUBTYPE OF (maths_space, generic_literal);
  min : INTEGER;
END_ENTITY;
(*
```

Attribute definitions:

min: The smallest integer in the interval.

4.5.23 integer_interval_to_max

This type of **maths_space** and **generic_literal** represents semi-infinite integer intervals which contain all integers less than or equal to a given integer.

EXPRESS specification:

```
*)
ENTITY integer_interval_to_max
  SUBTYPE OF (maths_space, generic_literal);
  max : INTEGER;
END_ENTITY;
(*
```

Attribute definitions:

max: The largest integer in the interval.

EXAMPLE 1 The space of all negative integers may be represented by **maths_space** () || **generic_expression** () || **simple_generic_expression** () || **generic_literal** () || **integer_interval_to_max** (-1).

4.5.24 finite_real_interval

This type of **maths_space** and **generic_literal** represents mathematical spaces which are intervals of the real numbers having finite positive length.

© ISO 2002 — All rights reserved

NOTE The finiteness of these intervals is exhibited in their lengths, not the number of members.

EXPRESS specification:

```
*)
ENTITY finite_real_interval
  SUBTYPE OF (maths_space, generic_literal);
  min         : REAL;
  min_closure : open_closed;
  max         : REAL;
  max_closure : open_closed;
WHERE
  WR1: min < max;
END_ENTITY;
(*
```

Attribute definitions:

min: The lower bound of the interval.

min_closure: Indicator for whether the lower bound is excluded (**open**) or included (**closed**) in the interval.

max: The upper bound of the interval.

max_closure: Indicator for whether the upper bound is excluded (**open**) or included (**closed**) in the interval.

Formal propositions:

WR1: The lower bound shall be strictly less than the upper bound.

4.5.25 real_interval_from_min

This type of **maths_space** and **generic_literal** represents mathematical spaces which are intervals of real numbers which are bounded below but not bounded above. That is, it represents intervals which contain all real numbers either greater than, or greater than or equal to, a given real number.

EXPRESS specification:

```
*)
ENTITY real_interval_from_min
  SUBTYPE OF (maths_space, generic_literal);
  min         : REAL;
  min_closure : open_closed;
END_ENTITY;
(*
```

 © ISO 2002 — All rights reserved

Attribute definitions:

min: The lower bound for the interval.

min_closure: Indicator for whether the lower bound is excluded (**open**) or included (**closed**) in the interval.

4.5.26 real_interval_to_max

This type of **maths_space** and **generic_literal** represents mathematical spaces which are intervals of real numbers which are bounded above but not bounded below. That is, it represents intervals which contain all real numbers either less than, or less than or equal to, a given real number.

EXPRESS specification:

```
*)
ENTITY real_interval_to_max
  SUBTYPE OF (maths_space, generic_literal);
  max         : REAL;
  max_closure : open_closed;
END_ENTITY;
(*
```

Attribute definitions:

max: The upper bound for the interval.

max_closure: Indicator for whether the upper bound is excluded (**open**) or included (**closed**) in the interval.

4.5.27 cartesian_complex_number_region

This type of **maths_space** and **generic_literal** represents subsets of the complex numbers defined by restricting the real parts to a specified, non-trivial, real interval and the imaginary parts to another such interval.

EXPRESS specification:

```
*)
ENTITY cartesian_complex_number_region
  SUBTYPE OF (maths_space, generic_literal);
  real_constraint : real_interval;
  imag_constraint : real_interval;
WHERE
  WR1: min_exists(real_constraint) OR max_exists(real_constraint) OR
       min_exists(imag_constraint) OR max_exists(imag_constraint);
END_ENTITY;
(*
```

© ISO 2002 — All rights reserved

Attribute definitions:

real_constraint: The interval to which the real parts of the members shall belong.

imag_constraint: The interval to which the imaginary parts of the members shall belong.

Formal propositions:

WR1: This type shall not be used to represent the space of all complex numbers.

4.5.28 polar_complex_number_region

This type of **maths_space** and **generic_literal** represents subsets of the complex numbers defined by restricting the distance and direction from a designated centre to specified, non-trivial, real intervals. Direction shall be measured in radians from the ray emanating from the designated centre parallel to the positive real axis, using values θ in the range $-\pi < \theta \leq \pi$, and using positive values for directions tending upward. The direction θ of a complex number from the designated centre shall be deemed to belong to a real interval if either θ or $\theta + 2\pi$ is a member of the interval.

EXPRESS specification:

```
*)
ENTITY polar_complex_number_region
  SUBTYPE OF (maths_space, generic_literal);
  centre               : complex_number_literal;
  distance_constraint  : real_interval;
  direction_constraint : finite_real_interval;
WHERE
  WR1: min_exists(distance_constraint) AND (real_min(distance_constraint) >= 0.0);
  WR2: {-PI <= direction_constraint.min < PI};
  WR3: direction_constraint.max - direction_constraint.min <= 2.0*PI;
  WR4: (direction_constraint.max - direction_constraint.min < 2.0*PI) OR
       (direction_constraint.min_closure = open);
  WR5: (direction_constraint.max - direction_constraint.min < 2.0*PI) OR
       (direction_constraint.max_closure = open) OR
       (direction_constraint.min = -PI);
  WR6: (real_min(distance_constraint) > 0.0) OR max_exists(distance_constraint) OR
       (direction_constraint.max - direction_constraint.min < 2.0*PI) OR
       (direction_constraint.max_closure = open);
END_ENTITY;
(*
```

Attribute definitions:

centre: The designated centre.

distance_constraint: The interval constraining the distances of the members from the centre.

direction_constraint: The interval constraining the directions of the members from the centre.

 © ISO 2002 — All rights reserved

Formal propositions:

WR1: The real interval defining the distance constraint shall have a greatest lower bound and it shall be nonnegative.

WR2: The lower bound of the real interval defining the direction constraint shall be greater than or equal to $-\pi$ and less than π.

WR3: The length of the interval defining the direction constraint shall be less than or equal to two π.

WR4: If the length of the interval defining the direction constraint is two π, then the interval shall be open below.

WR5: If all directions are included, then the interval defining the direction constraint shall start at $-\pi$.

WR6: This type shall not be used to represent the space of all complex numbers.

NOTE 1 Direction is naturally defined only up to an integer multiple of 2π. In order to represent *any* arc of directions via a *single* interval, a little of this inherent redundancy must be permitted and handled. Thus, the range of possible values for **direction_constraint.max** will be seen to be $-\pi$ to 3π.

NOTE 2 The effect of rules **WR4**, **WR5**, and **WR6** is to establish value unique representations for the complex number regions representable by this type. That is, two entity instances from this part of ISO 10303 representing regions of complex numbers represent the same mathematical region if and only if they are value equal as EXPRESS instances.

4.5.29 finite_space

This type of **maths_space** and **generic_literal** represents mathematical spaces with a finite number of members. This includes the space with zero members, also known as the empty set.

EXPRESS specification:

```
*)
ENTITY finite_space
  SUBTYPE OF (maths_space, generic_literal);
  members : SET OF maths_value;
WHERE
  WR1: VALUE_UNIQUE(members);
  WR2: SIZEOF (QUERY (expr <* QUERY (member <* members |
       'ISO13584_GENERIC_EXPRESSIONS_SCHEMA.GENERIC_EXPRESSION' IN TYPEOF (member))
       | NOT expression_is_constant(expr))) = 0;
  WR3: no_cyclic_space_reference(SELF, []);
END_ENTITY;
(*
```

Attribute definitions:

members: The set of members belonging to the space represented.

© ISO 2002 — All rights reserved

Formal propositions:

WR1: The members are distinct with respect to value equality.

WR2: All the members which are expressions are constant expressions.

WR3: The graph of **maths_space** references emanating from an instance of **finite_space** contains no cycles.

Informal propositions:

IP1: The members shall denote distinct mathematical values, for example, the number "4" and the expression "2+2" shall not both be members of the same instance of **finite_space**.

4.5.30 uniform_product_space

This type of **maths_space** and **generic_literal** represents finite Cartesian product spaces, all of whose factor spaces are the same. These Cartesian products are commonly written using an exponential notation. Let B be the space represented by the **base** attribute and n be the value of the **exponent** attribute. Then the space represented by the instance is commonly expressed as B^n.

NOTE There is a unique, natural isomorphism between the space B and the space B^1. It is a very common practice in mathematics to identify these two spaces via this isomorphism and not distinguish between elements of B and the corresponding one-tuples in B^1. In this schema, this distinction must be maintained in order to avoid computer processing ambiguity in later applications.

EXPRESS specification:

```
*)
ENTITY uniform_product_space
  SUBTYPE OF (maths_space, generic_literal);
  base     : maths_space;
  exponent : positive_integer;
WHERE
  WR1: expression_is_constant(base);
  WR2: no_cyclic_space_reference(SELF, []);
  WR3: base <> the_empty_space;
END_ENTITY;
(*
```

Attribute definitions:

base: The **maths_space** to which the components of the ordered tuples belong, that is, the space which is "multiplied" together **exponent** times to form the Cartesian product space.

exponent: The number of factors in the Cartesian product, which is also the length of the ordered tuples belonging to the space being represented.

 © ISO 2002 — All rights reserved

Formal propositions:

WR1: The value of **base** is a constant expression.

WR2: The graph of **maths_space** references emanating from an instance of **uniform_product_space** contains no cycles.

WR3: The value of **base** shall not be the empty space.

4.5.31 listed_product_space

This type of **maths_space** and **generic_literal** represents finite Cartesian product spaces. Its use is required when the factor spaces are not all the same. Let S_i be the space represented by the ith member of the attribute **factors**, and let the length of the list be n. The space represented by the instance is

$$\times_{i=1}^{n} S_i$$

NOTE The spaces represented by instances of **uniform_product_space** are also representable by instances **listed_product_space**. Since redundant representation is inherent in mathematical expressions and in the possibility of multiple instances of the same entity type representing the same mathematical object, no advantage was seen in attempting to add rules to prevent it or in forcing use of an inconvenient representation. Consider representing R^{64} using an instance of **listed_product_space**.

EXPRESS specification:

```
*)
ENTITY listed_product_space
  SUBTYPE OF (maths_space, generic_literal);
  factors : LIST OF maths_space;
WHERE
  WR1: SIZEOF (QUERY (space <* factors |
       NOT (expression_is_constant(space)))) = 0;
  WR2: no_cyclic_space_reference(SELF, []);
  WR3: NOT (the_empty_space IN factors);
END_ENTITY;
(*
```

Attribute definitions:

factors: The list of mathematical spaces which are the factors of the finite Cartesian product space.

Formal propositions:

WR1: The members of **factors** are all constant expressions.

WR2: The graph of **maths_space** references emanating from an instance of **listed_product_space** contains no cycles.

WR3: The empty space shall not be a factor space.

© ISO 2002 — All rights reserved

EXAMPLE The mathematical space of all triples such that the first component is an integer, the second is a real number in the closed interval from minus one to one, and the third is a boolean value:

$$Z \times \{x \in R | -1 \leq x \leq 1\} \times \{true, false\}$$

where Z represents the set of integers, can be represented by the entity instance:

```
LOCAL
  spc : listed_product_space := make_listed_product_space([the_integers,
    the_neg1_one_interval, the_booleans]);
END_LOCAL;
```

Three constants representing the needed factor spaces are defined in the **mathematical_functions_-schema** and used here to make one new instance representing the desired Cartesian product space.

4.5.32 extended_tuple_space

This type of **maths_space** and **generic_literal** represents spaces of ordered tuples with a certain number of initial components belonging to a fixed list of spaces, followed by an arbitrary number of components from a fixed space. These spaces represent domains of functions with a fixed list of initial arguments and types followed by an arbitrary number of additional arguments, all of the same type. The fixed list of initial arguments may be an empty list.

NOTE Among functions with an indefinite number of inputs, this is the only type provided for in this version of this schema.

The tuple spaces represented by this type can be expressed as infinite unions of finite Cartesian products of a certain simple form. Let B be the finite Cartesian product space represented by the value of **base** and E be the space represented by the value of **extender**. Then the space represented by the instance of **extended_tuple_space** is

$$\bigcup_{i=0}^{\infty} B \times_a E^i$$

where the associative Cartesian product operation is being used.

EXPRESS specification:

```
*)
ENTITY extended_tuple_space
  SUBTYPE OF (maths_space, generic_literal);
  base     : product_space;
  extender : maths_space;
WHERE
  WR1: expression_is_constant(base) AND
       expression_is_constant(extender);
  WR2: no_cyclic_space_reference(SELF, []);
  WR3: extender <> the_empty_space;
END_ENTITY;
(*
```

Attribute definitions:

base: The **product_space** describing the common initial component spaces of all the ordered tuples belonging to this tuple space. When there are no common initial components, the value

 © ISO 2002 — All rights reserved

of **base** will be the zero-tuple space.

extender: The **maths_space** from which the optional additional components come.

Formal propositions:

WR1: The values of **base** and **extender** are constant expressions.

WR2: The graph of **maths_space** references emanating from an instance of **extended_tuple_space** contains no cycles.

WR3: The value of **extender** shall not be the empty space.

4.5.33 function_space

This type of **maths_space** and **generic_literal** represents mathematical spaces whose members are all the mathematical functions which satisfy the indicated constraints on their domains and ranges.

EXPRESS specification:

```
*)
ENTITY function_space
  SUBTYPE OF (maths_space, generic_literal);
  domain_constraint : space_constraint_type;
  domain_argument   : maths_space;
  range_constraint : space_constraint_type;
  range_argument    : maths_space;
WHERE
  WR1: expression_is_constant(domain_argument) AND
       expression_is_constant(range_argument);
  WR2: (domain_argument <> the_empty_space) AND
       (range_argument <> the_empty_space);
  WR3: (domain_constraint <> sc_member) OR NOT
       member_of(the_empty_space,domain_argument);
  WR4: (range_constraint <> sc_member) OR NOT
       member_of(the_empty_space,range_argument);
  WR5: NOT (any_space_satisfies(domain_constraint,domain_argument) AND
       any_space_satisfies(range_constraint,range_argument));
END_ENTITY;
(*
```

Attribute definitions:

domain_constraint: The enumeration item identifying the kind of constraint to be imposed on the function domain.

domain_argument: The mathematical space to be used in the constraint identified by **domain_constraint**.

range_constraint: The enumeration item identifying the kind of constraint to be imposed on the function range.

range_argument: The mathematical space to be used in the constraint identified by **range_-**

© ISO 2002 — All rights reserved

constraint.

Formal propositions:

WR1: The values of **domain_argument** and **range_argument** shall be constant expressions.

WR2: The values of **domain_argument** and **range_argument** shall not represent the empty space.

WR3: If the **domain_constraint** is **sc_member**, then the empty space shall not be a member of the **domain_argument**.

WR4: If the **range_constraint** is **sc_member**, then the empty space shall not be a member of the **range_argument**.

WR5: This type shall not be used to represent the space of all mathematical functions.

NOTE The effect of rules **WR2**, **WR3** and **WR4** is to prevent instances of this type from representing the empty set or the set whose only member is the empty function.

4.5.34 maths_function

This abstract supertype is the all-inclusive type for representations of mathematical functions.

The derived **domain** attribute describes the number and kinds of inputs to which the function may be applied. The derived **range** attribute describes the number and kinds of outputs which the function may produce.

EXPRESS specification:

```
*)
ENTITY maths_function
  ABSTRACT SUPERTYPE OF (ONEOF (finite_function,
                                constant_function,
                                selector_function,
                                elementary_function,
                                restriction_function,
                                repackaging_function,
                                reindexed_array_function,
                                series_composed_function,
                                parallel_composed_function,
                                explicit_table_function,
                                homogeneous_linear_function,
                                general_linear_function,
                                b_spline_basis,
                                b_spline_function,
                                rationalize_function,
                                partial_derivative_function,
                                definite_integral_function,
                                abstracted_expression_function,
                                expression_denoted_function,
                                imported_point_function,
                                imported_curve_function,
                                imported_surface_function,
                                imported_volume_function,
```


© ISO 2002 — All rights reserved

```
                                application_defined_function))
  SUBTYPE OF (generic_expression);
DERIVE
  domain : tuple_space := derive_function_domain(SELF);
  range : tuple_space := derive_function_range(SELF);
END_ENTITY;
(*
```

Attribute definitions:

domain: The derived tuple space whose members are all the ordered tuples of inputs to which this function may properly be applied.

range: The derived tuple space whose members are to be considered possible ordered tuples of outputs of this function. No tuple of outputs of this function other than "?" may fail to be a member of this space.

NOTE 1 The values of the **domain** and **range** attributes constrain the roles that an instance of **maths_function** may fill in other data structures.

NOTE 2 These derived attributes are the nearest EXPRESS equivalent of the pure virtual member functions of an object-oriented programming language such as C++.

NOTE 3 Most functions have a single output. However, for reasons of symmetry, and also to permit the modeling of "procedures" which return multiple outputs through the use of "VAR" arguments, the possibility of multiple outputs is permitted. In fact, the possibility of functions producing an indefinite number of outputs is permitted. (Consider the scanf() function in the C programming language.)

NOTE 4 See also the note about dual inheritance in 4.5.19.

4.5.35 finite_function

This type of **maths_function** and **generic_literal** provides representations for functions specified by simply collecting all pairs of input and corresponding output values.

The domain and range are derived from the value of **pairs** by constructing the **finite_space** of all first members and all second members, respectively.

EXPRESS specification:

```
*)
ENTITY finite_function
  SUBTYPE OF (maths_function, generic_literal);
  pairs : SET [1:?] OF LIST [2:2] OF maths_value;
WHERE
  WR1: VALUE_UNIQUE(list_selected_components(pairs, 1));
END_ENTITY;
(*
```

Attribute definitions:

pairs: The complete set of pairs of input and output values.

© ISO 2002 — All rights reserved

Formal propositions:

WR1: No two elements of **pairs** may have the same first member.

EXAMPLE Suppose one wishes to represent a function mapping the three strings "sin", "cos", and "tan" to the corresponding trigonometric functions. This can be accomplished using an instance of **finite_function** as follows:

```
LOCAL
  sine_f : elementary_function := make_elementary_function(ef_sin_r);
  cosine_f : elementary_function := make_elementary_function(ef_cos_r);
  tangent_f : elementary_function := make_elementary_function(ef_tan_r);
  translator_f : finite_function := make_finite_function([['sin',sine_f],
    ['cos',cosine_f],['tan',tangent_f]]);
END_LOCAL;
```

4.5.36 constant_function

This type of **maths_function** and **generic_literal** provides representations for functions producing the same output for any input.

The range of a **constant_function** is derived to be the space of one-tuples of the space whose only element is the object represented by the value of **sole_output**.

The domain of a **constant_function** is derived from the value *src* of **source_of_domain** as follows:

— If *src* is a **tuple_space**, *src* is the function domain.

— Else if *src* is a **maths_space**, the space of one-tuples from *src* is the function domain.

— Else *src* is a **maths_function** and the domain of *src* is the constant function domain.

NOTE The choice of references from which the domain of a **constant_function** can be derived reflects the fact that constant functions are usually created to serve some minor role in a larger construct, and that larger construct usually determines the desired domain for the constant function. In particular, function domains are often derived rather than explicitly represented, and this mechanism allows a constant function to "borrow" that derivation from another function.

EXPRESS specification:

```
*)
ENTITY constant_function
  SUBTYPE OF (maths_function, generic_literal);
  sole_output      : maths_value;
  source_of_domain : maths_space_or_function;
WHERE
  WR1: no_cyclic_domain_reference(source_of_domain, [SELF]);
  WR2: expression_is_constant(domain_from(source_of_domain));
END_ENTITY;
(*
```

 © ISO 2002 — All rights reserved

Attribute definitions:

sole_output: The value representing the only valid output produced by this function.

source_of_domain: The object from which the function domain shall be derived.

Formal propositions:

WR1: The chain of references leading to the function domain shall not contain any cycles.

WR2: The expression obtained for the domain shall not contain free variables, that is, it shall denote a single, well-defined, mathematical space.

4.5.37 selector_function

This type of **maths_function** and **generic_literal** provides representations for functions producing an output by selecting one of the inputs or, when there is only one input and it is a tuple, by selecting one component of that sole input.

The function domain of a **selector_function** is derived from the value *src* of **source_of_domain** as follows:

— If *src* is a **tuple_space**, *src* is the function domain.

— Else if *src* is a **maths_space**, the space of one-tuples from *src* is the function domain.

— Else *src* is a **maths_function** and the domain of *src* is the selector function domain.

The range of a **selector_function** is derived to be the space of one-tuples of the appropriate component of the domain.

NOTE The choice of references from which the domain of a **selector_function** can be derived reflects the fact that selector functions are usually created to serve some minor role in a larger construct, and that larger construct usually determines the desired domain for the selector function. In particular, function domains are often derived rather than explicitly represented, and this mechanism allows a selector function to "borrow" that derivation from another function.

EXPRESS specification:

```
*)
ENTITY selector_function
  SUBTYPE OF (maths_function, generic_literal);
  selector : input_selector;
  source_of_domain : maths_space_or_function;
WHERE
  WR1: no_cyclic_domain_reference(source_of_domain, [SELF]);
  WR2: expression_is_constant(domain_from(source_of_domain));
END_ENTITY;
(*
```

© ISO 2002 — All rights reserved

Attribute definitions:

selector: The value indicating which input or which component of the first input shall be produced as the output. See 4.4.14.

source_of_domain: The object from which the function domain shall be derived.

Formal propositions:

WR1: The chain of references leading to the function domain shall not contain any cycles.

WR2: The expression obtained for the domain shall not contain free variables, that is, it shall denote a single, well-defined, mathematical space.

4.5.38 elementary_function

This type of **maths_function** and **generic_literal** provides representations for the elementary operators, functions and procedures identified by **elementary_function_enumerators** items.

The domain and range are derived from the value of **func_id**.

EXPRESS specification:

```
*)
ENTITY elementary_function
  SUBTYPE OF (maths_function, generic_literal);
  func_id : elementary_function_enumerators;
END_ENTITY;
(*
```

Attribute definitions:

func_id: The enumeration value identifying the elementary function, procedure or operation being represented.

4.5.39 restriction_function

This type of **maths_function** and **unary_generic_expression** represents identity functions on given mathematical spaces. They may be used to restrict the effective domain or range of another function by composing them with the other function in an appropriate order.

Restriction functions take a single value from the space indicated by **operand** as input and produce the same single value as output. The domain and range are derived accordingly to be the spaces of all one-tuples from the space identified by the **operand** value.

EXPRESS specification:

```
*)
ENTITY restriction_function
```

 © ISO 2002 — All rights reserved

```
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_space;
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be an instance of **maths_space** and identifies the mathematical space which will serve as the domain and the range of the **restriction_function**.

4.5.40 repackaging_function

This type of **maths_function** and **unary_generic_expression** represents simple variations of another function differing only in the "packaging" of its inputs or outputs. Specifically, a function taking multiple inputs can be repackaged to take a single input which is the tuple of the original inputs, and vice versa. Similarly a function producing multiple outputs can be repackaged to produce a single output which is the tuple of the original outputs, or vice versa. Finally, a function producing multiple outputs can be repackaged to be the function producing just one of those outputs, the others being discarded.

The domain and range are derived from those of the function indicated by **operand**, with the indicated changes in packaging.

NOTE Formally, all these manipulations can also be represented as compositions of the original function with appropriate entuple, detuple, and tuple component selection functions. The **repackaging_function** makes representation of these common manipulations much easier and more efficient.

The need to perform such manipulations arises from the schema's need to maintain distinctions which are not maintained in informal mathematics. Informal mathematics often does not distinguish between a space X and the space of all one tuples of elements of X, which is X^1. Nor does it usually distinguish carefully between a function taking three real inputs and a function taking one input which is a triple of reals. See 4.2.6 and 4.2.7.

EXPRESS specification:

```
*)
ENTITY repackaging_function
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
  input_repack    : repackage_options;
  output_repack   : repackage_options;
  selected_output : nonnegative_integer;
WHERE
  WR1: (input_repack <> ro_wrap_as_tuple) OR
       ((space_dimension(operand.domain) = 1) AND
         ((schema_prefix + 'TUPLE_SPACE') IN TYPEOF (factor1(operand.domain))));
  WR2: (output_repack <> ro_unwrap_tuple) OR
       ((space_dimension(operand.range) = 1) AND
         ((schema_prefix + 'TUPLE_SPACE') IN TYPEOF (factor1(operand.range))));
  WR3: selected_output <= space_dimension( repackage(
       operand.range, output_repack));
END_ENTITY;
```

© ISO 2002 — All rights reserved

```
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be an instance of **maths_function** and identifies the function to be repackaged, herinafter referred to as the original function.

input_repack: The enumeration item indicating the repackaging operation to perform on the inputs before applying the original function.

output_repack: The enumeration item indicating the repackaging operation to perform on the outputs after applying the original function.

selected_output: If positive, the output which is selected from the output of the **output_-repack** operation to be the sole output of the repackaged function. If zero, the output is that resulting from **output_repack**.

Formal propositions:

WR1: If the input repackaging option is **ro_wrap_as_tuple**, then the function to be repackaged shall take a single input which is a tuple object.

WR2: If the output repackaging option is **ro_unwrap_tuple**, then the function to be repackaged shall produce a single output which is a tuple object.

WR3: The value of **selected_output** shall be less than or equal to the number of outputs after the effect of **output_repack** is taken into account.

EXAMPLE 1 In the example of a **rationalize_function**, the function `rf` takes a one-tuple of real numbers as its input. If it were desired to represent the corresponding function which takes a single real number as its input, this could be accomplished using an instance of **repackaging_function** as follows:

```
LOCAL
  rf_variant : repackaging_function := make_repackaging_function(rf, ro_wrap_as_tuple,
    ro_nochange, 0);
END_LOCAL;
```

EXAMPLE 2 Suppose that in the previous example the function `rf_variant` produces a single output value which is a triple of real numbers. If one wished to represent the function which produced three outputs, each a real number, one could proceed as follows:

```
LOCAL
  rf_variant2 : repackaging_function := make_repackaging_function(rf, ro_wrap_as_tuple,
    ro_unwrap_tuple, 0);
END_LOCAL;
```

Such a function would be called a procedure in most programming languages. In most programming languages, it would not be possible to use a procedure in an expression. In this Part, it is permitted, but the function shall be interpreted as denoting its first output only, the other outputs being irrelevant to the interpretation of such an expression.

EXAMPLE 3 As a final example, suppose one wished to have the function of the previous example denote its third output. There are several ways a "procedure" can be converted into a proper function, but the simplest is to make use of the **selected_output** attribute of the **repackaging_function** as follows:

 © ISO 2002 — All rights reserved

```
LOCAL
  rf_variant3 : repackaging_function := make_repackaging_function(rf, ro_wrap_as_tuple,
    ro_unwrap_tuple, 3);
END_LOCAL;
```

4.5.41 reindexed_array_function

This type of **maths_function** and **unary_generic_expression** represents the array function corresponding to to a given array function but having shifted indexing intervals. The indexing for the array function being represented is specified by giving the starting indices for each subscript.

The domain of the **reindexed_array_function** is derived to match that of the value of the inherited **operand** attribute except that each finite integer interval component is shifted to have the indicated starting index.

The range of the **reindexed_array_function** is derived to be the same as that of the value of the inherited **operand** attribute.

NOTE In most cases, the **operand** value will be a table function represented using one of the other subtypes of **maths_function**. The present type serves as a convenient mechanism for representing arrays whose indices don't all start at zero or at one.

EXPRESS specification:

```
*)
ENTITY reindexed_array_function
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
  starting_indices : LIST [1:?] OF INTEGER;
WHERE
  WR1: function_is_array(SELF\unary_generic_expression.operand);
  WR2: SIZEOF(starting_indices) = SIZEOF(shape_of_array(
       SELF\unary_generic_expression.operand));
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** value identifies the array function to be reindexed.

starting_indices: The starting indices in each dimension of the domain of the represented array function.

Formal propositions:

WR1: The inherited **operand** value shall be an array function.

WR2: The size of the list of starting indices shall match the number of dimensions of the array function being reindexed.

EXAMPLE See example 3 in 4.5.53.

© ISO 2002 — All rights reserved

4.5.42 series_composed_function

This type of **maths_function** and **multiple_arity_generic_expression** represents the mathematical function which is the serial composition of the functions in its inherited **operands** attribute. The output of an instance of **series_composed_function** is obtained from its input by applying **operands[1]** to its input, **operands[2]** to the output from **operands[1]**, and so on, until the output of the **composed_function** is obtained as the output of the last member of **operands**. In mathematical notation, the **series_composed_function** F is constructed from the list of functions $f_1, f_2, f_3, ..., f_n$ indicated by its **operands** attribute value by $F(x) \equiv f_n(...f_3(f_2(f_1(x)))...)$, where x represents any list of actual inputs.

The domain of the **series_composed_function** is derived to be the domain of its first operand and its range is derived as that of its last operand.

EXPRESS specification:

```
*)
ENTITY series_composed_function
  SUBTYPE OF (maths_function, multiple_arity_generic_expression);
  SELF\multiple_arity_generic_expression.operands : LIST [2:?] of maths_function;
WHERE
  WR1: composable_sequence(SELF\multiple_arity_generic_expression.operands);
END_ENTITY;
(*
```

Attribute definitions:

SELF\multiple_arity_generic_expression.operands: The inherited **operands** attribute is constrained to be a list of **maths_function** instances and identifies the functions to be composed and the order of their application.

Formal propositions:

WR1: The outputs of a function in the sequence shall be suitable as inputs to the next function in the sequence.

4.5.43 parallel_composed_function

This type of **maths_function** and **multiple_arity_generic_expression** represents the mathematical function which is the composition of a "final" function with a list of "preparatory" functions supplying its arguments. The output of an instance of **parallel_composed_function** is obtained from its input by applying each of its preparatory functions in parallel to its input, and applying the final function to the collected outputs of the preparatory functions. In mathematical notation, the **parallel_composed_function** F is constructed from a final function g and a list of preparatory functions $f_1, f_2, f_3, ..., f_n$ by $F(x) \equiv g(f_1(x), f_2(x), f_3(x), ..., f_n(x))$, where x represents any list of actual inputs.

The domain of a **parallel_composed_function** is derived from the value *src* of **source_of_-domain** as follows:

 © ISO 2002 — All rights reserved

— If *src* is a **tuple_space**, *src* is the function domain.

— Else if *src* is a **maths_space**, the space of one-tuples from *src* is the function domain.

— Else *src* is a **maths_function** and the domain of *src* is the function domain.

The range of the **parallel_composed_function** is derived to be the range of the value of its **final_function** attribute.

EXPRESS specification:

```
*)
ENTITY parallel_composed_function
  SUBTYPE OF (maths_function, multiple_arity_generic_expression);
  source_of_domain  : maths_space_or_function;
  prep_functions : LIST [1:?] OF maths_function;
  final_function : maths_function_select;
DERIVE
  SELF\multiple_arity_generic_expression.operands : LIST [2:?] of generic_expression
    := convert_to_operands_prcmfn(source_of_domain, prep_functions, final_function);
WHERE
  WR1: no_cyclic_domain_reference(source_of_domain, [SELF]);
  WR2: expression_is_constant(domain_from(source_of_domain));
  WR3: parallel_composed_function_domain_check(domain_from(source_of_domain),
       prep_functions);
  WR4: parallel_composed_function_composability_check(prep_functions, final_function);
END_ENTITY;
(*
```

Attribute definitions:

source_of_domain: The source of the domain of the **parallel_composed_function**.

prep_functions: The list of preparatory functions.

final_function: The final function.

SELF\multiple_arity_generic_expression.operands: The inherited **operands** attribute is derived to be a list obtained by concatenating the values of the three explicit attributes in the given order, generating instances of **generic_literal** as necessary.

Formal propositions:

WR1: The chain of references leading to the function domain shall not contain any cycles.

WR2: The expression obtained for the domain shall not contain free variables, that is, it shall denote a single, well-defined, mathematical space.

WR3: The domains of all the preparatory functions shall be compatible with the domain of the function.

WR4: The associative Cartesian product of the ranges of the preparatory functions shall be compatible with the domain of the final function.

© ISO 2002 — All rights reserved

NOTE 1 Ideally, the domain of a **parallel_composed_function** would be derived as the intersection of the domains of its preparatory functions. Since this is not readily computable from the information at hand, an explicit domain attribute is provided and it is required that the domains of the preparatory functions be compatible with it. Recall that in the context of this part of ISO 10303, two spaces are compatible if they are not known to have empty intersection.

NOTE 2 The reason for using the associative Cartesian product in the **WR4** rule is somewhat subtle. Consider two preparatory functions, each of which produces a single output which is a triple of real numbers. An appropriate final function must take two inputs, each of which is a triple of real numbers. The two range spaces are, formally, $(R^3)^1$. The Cartesian product of these two spaces is $((R^3)^1)^2$, but the associative Cartesian product is $(R^3)^2$, as desired.

EXAMPLE 1 Suppose one has a function $srf(u,v) = (x,y,z)$, represented by an instance `"srf"` of **maths_function** and a displacement function $disp(u,v) = (dx,dy,dz)$, defined on the same parametric domain and represented by another instance `"disp"` of **maths_function** and one wishes to represent the sum of these functions, $sum(u,v) = srf(u,v) + disp(u,v)$. An instance of **parallel_composed_function** represents the sum as follows:

```
LOCAL
  sum : parallel_composed_function := make_parallel_composed_function(
          srf, [srf, disp], ef_add_rt);
END_LOCAL;
```

Observe that the nominal input and output variables used in the informal mathematical description played no essential role and need not be represented. That is, knowing that srf and $disp$ are functions with the same domain and compatible ranges is sufficient to permit the representation of $sum \equiv srf + disp$.

EXAMPLE 2 A more challenging example is to represent $movsrf(u,v,t) \equiv srf(u,v) + 2 \cdot t \cdot disp(u,v)$. The problem is that the natural domains of the component functions differ. One solution is to replace the natural components with equivalent functions which are defined on the required common domain. Define input selection functions $selu(u,v,t) \equiv (u)$, $selv(u,v,t) \equiv (v)$, and $selt(u,v,t) \equiv (t)$, and the constant function $con2(u,v,t) \equiv (2.0)$. Then $movsrf \equiv srf(selu, selv) + con2 \cdot selt \cdot disp(selu, selv)$, can be represented using five instances of **parallel_composed_function** as follows.

```
LOCAL
  comdom  : tuple_space := assoc_product_space(srf\maths_function.domain, the_reals);
  selu    : selector_function := make_selector_function(1, comdom);
  selv    : selector_function := make_selector_function(2, comdom);
  selt    : selector_function := make_selector_function(3, comdom);
  con2    : constant_function := make_constant_function(2.0, comdom);
  altsrf  : parallel_composed_function := make_parallel_composed_function(
              comdom, [selu, selv], srf);
  altdisp : parallel_composed_function := make_parallel_composed_function(
              comdom, [selu, selv], disp);
  two_t   : parallel_composed_function := make_parallel_composed_function(
              comdom, [con2, selt], ef_multiply_r);
  prod    : parallel_composed_function := make_parallel_composed_function(
              comdom, [two_t, altdisp], ef_scalar_mult_rt);
  movsrf  : parallel_composed_function := make_parallel_composed_function(
              comdom, [altsrf, prod], ef_add_rt);
END_LOCAL;
```

EXAMPLE 3 If the variables u, v and t in the preceding example actually have significance to the application, then an alternate representation containing explicit instances representing these variables might be preferred. Such a representation could be constructed as follows.

```
LOCAL
  udom   : maths_space := factor1(srf.domain);
```

 © ISO 2002 — All rights reserved

```
  vdom   : maths_space := factor_space(srf.domain, 2);
  sem    : bound_variable_semantics := bound_variable_semantics() ||
             variable_semantics();
  u      : maths_real_variable := make_maths_real_variable(udom, 'u');
  u_env  : environment := make_environment(u, sem);
  v      : maths_real_variable := make_maths_real_variable(vdom, 'v');
  v_env  : environment := make_environment(v, sem);
  t      : maths_real_variable := make_maths_real_variable(the_reals, 't');
  t_env  : environment := make_environment(t, sem);
  two    : real_literal := make_real_literal(2.0);
  srfuv  : function_application := make_function_application(srf, [u,v]);
  dispuv : function_application := make_function_application(disp, [u,v]);
  two_t  : mult_expression := make_mult_expression([two,t]);
  prod   : function_application := make_function_application(ef_scalar_mult_rt,
             [two_t, dispuv]);
  expr   : function_application := make_function_application(ef_add_rt, [srfuv, prod]);
  movsrf : abstracted_expression_function := make_abstracted_expression_function(
             [expr, u, v, t]);
END_LOCAL;
```

If the expression $srf(u,v) + 2 \cdot t \cdot disp(u,v)$ is the object of communication and not the function $movsrf(u,v,t)$, then the construction of the instance representing the function can be omitted and **sem** replaced by one to three instances of **variable_semantics**, as appropriate, to describe the three variables as free variables and add whatever semantics may be necessary.

EXAMPLE 4 Define a function F by cases as

$$F(x) \equiv \left\{ \begin{array}{ll} G(x), & \text{for } x \,\varepsilon A; \\ H(x), & \text{otherwise.} \end{array} \right\}$$

Assume x ranges over some superspace D of A and that D, A, G, and H are represented by entity instances named by the corresponding lowercase letters. Then F is represented by

```
LOCAL
  selx : selector_function := make_selector_function(1, d);
  cona : constant_function := make_constant_function(a, d);
  xina : parallel_composed_function := make_parallel_composed_function(d,
           [selx, cona], ef_member_of);
  f    : parallel_composed_function := make_parallel_composed_function(d,
           [xina, g, h], ef_if);
END_LOCAL;
```

4.5.44 explicit_table_function

This type of **maths_function** is a supertype carrying some common attributes of a family of types explicitly representing table functions. See 4.2.9. A table function is functionally equivalent to a multi-dimensional array, all of whose subscript ranges start with the same value, which is either zero or one.

NOTE 1 Matrices, tensors, grids and meshes of all types are typically represented by, or composed mainly of, instances of **explicit_table_function**.

NOTE 2 Whenever the individual subscript ranges of a multi-dimensional array are nominal, that is, the relative positions of elements matter, but the integers used as subscripts are otherwise irrelevant, a table function is an appropriate representation. If ordinal position numbering is customary in an application area, all subscript ranges will start at one. If numbering relative to the first position is customary in an application area, all subscript ranges will start at zero. Since both are widely used, both are supported.

NOTE 3 Representations for mathematical table functions may also occur as instances of **series_-**

© ISO 2002 — All rights reserved

composed_function, **parallel_composed_function** and many other types. Such instances are implicit table functions and will be recognized by the function **function_is_table**.

EXPRESS specification:

```
*)
ENTITY explicit_table_function
  ABSTRACT SUPERTYPE OF (ONEOF (listed_real_data,
                                listed_integer_data,
                                listed_logical_data,
                                listed_string_data,
                                listed_complex_number_data,
                                listed_data,
                                externally_listed_data,
                                linearized_table_function,
                                basic_sparse_matrix))
  SUBTYPE OF (maths_function);
  index_base : zero_or_one;
  shape      : LIST [1:?] OF positive_integer;
END_ENTITY;
(*
```

Attribute definitions:

index_base: Indicator whether to start all subscript ranges from zero or from one.

shape: The sizes of the individual subscript ranges.

4.5.45 listed_real_data

This type of **explicit_table_function** and **generic_literal** represents mathematical functions which take a single integer as input and output the real number at a corresponding position in a list. The corresponding position is defined to be the input value plus one minus the value of **self\explicit_table_function.index_base**.

These functions take a single integer from the interval from **index_base** to **index_base** + SIZEOF(**values**) - 1 as input and produce a single real number as output.

NOTE This class of functions is functionally equivalent to one-dimensional tables of reals.

EXPRESS specification:

```
*)
ENTITY listed_real_data
  SUBTYPE OF (explicit_table_function, generic_literal);
  values : LIST [1:?] OF REAL;
DERIVE
  self\explicit_table_function.shape : LIST [1:?] OF positive_integer :=
    [SIZEOF (values)];
END_ENTITY;
(*
```

 © ISO 2002 — All rights reserved

Attribute definitions:

values: The list containing the output values of the function in order.

self\explicit_table_function.shape: The inherited shape attribute is derived to be the one-element list containing the number of values.

4.5.46 listed_integer_data

This type of **explicit_table_function** and **generic_literal** represents mathematical functions which take a single integer as input and output the integer at a corresponding position in a list. The corresponding position is defined to be the input value plus one minus the value of **self\explicit_table_function.index_base**.

These functions take a single integer from the interval from **index_base** to **index_base** + SIZEOF(**values**) - 1 as input and produce a single integer as output.

NOTE This class of functions is functionally equivalent to one-dimensional tables of integers.

EXPRESS specification:

```
*)
ENTITY listed_integer_data
  SUBTYPE OF (explicit_table_function, generic_literal);
  values : LIST [1:?] OF INTEGER;
DERIVE
  self\explicit_table_function.shape : LIST [1:?] OF positive_integer :=
    [SIZEOF (values)];
END_ENTITY;
(*
```

Attribute definitions:

values: The list containing the output values of the function in order.

self\explicit_table_function.shape: The inherited shape attribute is derived to be the one-element list containing the number of values.

EXAMPLE The following 24 integer values will be used as the underlying data set for a number of examples of table and array functions:

```
LOCAL
  rawdata : listed_integer_data := make_listed_integer_data(1,
    [111,211,311, 121,221,321, 131,231,331, 141,241,341,
     112,212,312, 122,222,322, 132,232,332, 142,242,342] );
END_LOCAL;
```

4.5.47 listed_logical_data

This type of **explicit_table_function** and **generic_literal** represents mathematical functions which take a single integer as input and output the EXPRESS logical value at a corresponding position in a list. The corresponding position is defined to be the input value plus one minus the value of **self\explicit_table_function.index_base**.

© ISO 2002 — All rights reserved

These functions take a single integer from the interval from **index_base** to **index_base** + SIZEOF(**values**) - 1 as input and produce a single EXPRESS logical value as output.

NOTE This class of functions is functionally equivalent to one-dimensional tables of EXPRESS logical values.

EXPRESS specification:

```
*)
ENTITY listed_logical_data
  SUBTYPE OF(explicit_table_function, generic_literal);
  values : LIST [1:?] OF LOGICAL;
DERIVE
  self\explicit_table_function.shape : LIST [1:?] OF positive_integer :=
    [SIZEOF (values)];
END_ENTITY;
(*
```

Attribute definitions:

values: The list containing the output values of the function in order.

self\explicit_table_function.shape: The inherited shape attribute is derived to be the one-element list containing the number of values.

4.5.48 listed_string_data

This type of **explicit_table_function** and **generic_literal** represents mathematical functions which take a single integer as input and output the string at a corresponding position in a list. The corresponding position is defined to be the input value plus one minus the value of **self\explicit_table_function.index_base**.

These functions take a single integer from the interval from **index_base** to **index_base** + SIZEOF(**values**) - 1 as input and produce a single string value as output.

NOTE This class of functions is functionally equivalent to one-dimensional tables of strings.

EXPRESS specification:

```
*)
ENTITY listed_string_data
  SUBTYPE OF (explicit_table_function, generic_literal);
  values : LIST [1:?] OF STRING;
DERIVE
  self\explicit_table_function.shape : LIST [1:?] OF positive_integer :=
    [SIZEOF (values)];
END_ENTITY;
(*
```

 © ISO 2002 — All rights reserved

Attribute definitions:

values: The list containing the output values of the function in order.

self\explicit_table_function.shape: The inherited shape attribute is derived to be the one-element list containing the number of values.

4.5.49 listed_complex_number_data

This type of **explicit_table_function** and **generic_literal** represents mathematical functions which take a single integer as input and output the complex number formed from two real numbers at a corresponding pair of consecutive positions in a list. Let the input value be j and the value of **self\explicit_table_function.index_base** be b. Then the first of the corresponding pair of consecutive positions in the list is $2(j-b)+1$.

These functions take a single integer from the interval from **index_base** to **index_base** + SIZEOF(**values**)/2 - 1 as input and produce a single complex number as output.

NOTE This class of functions is functionally equivalent to one-dimensional tables of complex numbers.

EXPRESS specification:

```
*)
ENTITY listed_complex_number_data
  SUBTYPE OF (explicit_table_function, generic_literal);
  values : LIST [2:?] OF REAL;
DERIVE
  self\explicit_table_function.shape : LIST [1:?] OF positive_integer :=
    [SIZEOF (values)/2];
WHERE
  WR1: NOT ODD (SIZEOF (values));
END_ENTITY;
(*
```

Attribute definitions:

values: The list containing the alternating real and imaginary parts of the output values of the function in order.

self\explicit_table_function.shape: The inherited shape attribute is derived to be the one-element list containing the number of **values** divided by two.

Formal propositions:

WR1: The size of the **values** list shall be even.

EXAMPLE Represent the one-dimensional table function of the four complex numbers: $2+3i, -4, i, e-\pi i$, indexed from zero.

```
LOCAL
  cx4 : listed_complex_number_data := make_listed_complex_number_data(0,
    [2.0, 3.0, -4.0, 0.0, 0.0, 1.0, CONST_E, PI]);
```

© ISO 2002 — All rights reserved

```
END_LOCAL;
```

4.5.50 listed_data

This type of **explicit_table_function** and **generic_literal** represents mathematical functions which take a single integer as input and output the **maths_value** at a corresponding position in a list. The corresponding position is defined to be the input value plus one minus the value of **self\explicit_table_function.index_base**.

These functions take a single integer from the interval from **index_base** to **index_base** + SIZEOF(**values**) - 1 as input and produce a single EXPRESS **maths_value** as output. The inherited **range** attribute is derived to be the space of one-tuples from **value_range**.

NOTE This class of functions is functionally equivalent to one-dimensional tables of **maths_value**s.

EXPRESS specification:

```
*)
ENTITY listed_data
  SUBTYPE OF (explicit_table_function, generic_literal);
  values      : LIST [1:?] OF maths_value;
  value_range : maths_space;
DERIVE
  SELF\explicit_table_function.shape : LIST [1:?] OF positive_integer :=
    [SIZEOF (values)];
WHERE
  WR1: expression_is_constant(value_range);
  WR2: SIZEOF (QUERY (val <* values | NOT (member_of( val, value_range)))) = 0;
END_ENTITY;
(*
```

Attribute definitions:

values: The list containing the output values of the function in order.

value_range: An instance of **maths_space** containing all the values in **values**.

SELF\explicit_table_function.shape: The inherited shape attribute is derived to be the one-element list containing the number of values.

Formal propositions:

WR1: The value of **value_range** shall be a constant expression.

WR2: The members of **values** shall all belong to the space **value_range**.

4.5.51 externally_listed_data

This type of **explicit_table_function**, **generic_literal** and **externally_defined_item** represents mathematical functions which take a single integer as input and output the **maths_value** at a corresponding position in some external, list-like, data source. The attributes inherited from **externally_defined_item** shall be used to identify the external data source. The nature

© ISO 2002 — All rights reserved

of the allowable external data sources, their identification, access, or exchange, and the precise definition of the correspondence between the input integer and values from the data source are not specified in this part of ISO 10303.

These functions take a single integer from the interval from **index_base** to **index_base** + **shape[1]** - 1 as input and produce a single EXPRESS **maths_value** belonging to **value_range** as output.

NOTE This class of functions is functionally equivalent to one-dimensional tables of **maths_value**s. One application of this type is for the case where the list of values is so large that embedding it in an exchange file is impractical. In such cases, the data values may be exchanged in a separate file in an especially efficient format, or maintained in a central respository which is only accessed as needed over a network. It is expected that the correspondence to a list-like structure will usually be obvious and implicit.

EXPRESS specification:

```
*)
ENTITY externally_listed_data
  SUBTYPE OF (explicit_table_function, generic_literal, externally_defined_item);
  value_range : maths_space;
WHERE
  WR1: expression_is_constant(value_range);
END_ENTITY;
(*
```

Attribute definitions:

value_range: An instance of **maths_space** containing all the values from the external data source which may appear as function outputs.

Formal propositions:

WR1: The value of **value_range** shall be a constant expression, that is, it shall denote one specific mathematical space.

4.5.52 linearized_table_function

This type of **explicit_table_function** and **unary_generic_expression** is a supertype carrying the common attributes of a family of entity types which make use of a one-dimensional array function to implement a many-dimensional table function. The common attributes supply the one-dimensional array function and an integer in the domain of the array function which is associated with a “first” or “base” position to be used by the table.

All the subtypes of this type represent classes of mathematical functions which take a single integer tuple as input and produce a single output value belonging to the sole factor space of the range of the one-dimensional array function. The output may actually be obtained by evaluating the one-dimensional array function or by using a default value specified in some other attribute of the subtype instance.

NOTE 1 This approach provides an efficient representation for cases where symmetry or sparsity exists

© ISO 2002 — All rights reserved

in the data or when many views (indexings) of all or parts of the same data set are required. In the former class of cases, **source** typically contains many fewer data values than the number of subscript tuples in the domain of the table. In the latter class of cases, **source** typically contains many more data values than are required for any one table, but many tables use the same **source** instance.

NOTE 2 In the most common usage, the **source** attribute will be an instance of one of the **listed_..._data** types. In the second most common usage, the **source** attribute will be an instance of **series_** or **parallel_composed_function**.

EXPRESS specification:

```
*)
ENTITY linearized_table_function
  SUPERTYPE OF (ONEOF (standard_table_function,
                       regular_table_function,
                       triangular_matrix,
                       symmetric_matrix,
                       banded_matrix))
  SUBTYPE OF (explicit_table_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
  first  : integer;
DERIVE
  source : maths_function := SELF\unary_generic_expression.operand;
WHERE
  WR1: function_is_1d_array(source);
  WR2: member_of(first, source.domain);
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be a **math_function** and provides the actual output value of the table function for non-defaulted cases.

first: The integer to be used as input to the function **source** to obtain the output value corresponding to the "first" subscript tuple, that is, the one created by using all zeros or all ones, depending on the inherited **index_base** attribute.

source: The short, convenient synonym for the inherited **operand** attribute. This is the source for independent individual values from which the output is produced. The indexing function used is determined by the subtype.

Formal propositions:

WR1: The value of **source** represents a one-dimensional array function.

WR2: The integer to be used to obtain the entry of the first position in the table shall be a member of the domain of **source**.

 © ISO 2002 — All rights reserved

4.5.53 standard_table_function

This type of **linearized_table_function** represents dense arrays of any dimension whose elements have been linearly ordered in one of two standard ways. A dense array is one whose structure supports an independent value for every subscript tuple in its domain. The two standard linear orderings for multi-dimensional arrays are described under the **ordering_type** type declaration.

The value associated with an input subscript tuple $[j_1, j_2, \ldots, j_n]$ is computed as follows: Let L be the function represented by the value of **SELF\linearized_table_function.source**, f be the value of **SELF\linearized_table_function.first**, s_i be the members of **SELF\explicit_table_function.shape**, and b be the value of **SELF\explicit_table_function.index_base**. Compute p_i as follows: If **ordering** is **by_rows**, let $p_i \equiv \prod_{k>i} s_k$; otherwise, let $p_i \equiv \prod_{k<i} s_k$. The table value for $[j_1, j_2, \ldots, j_n]$ is $L(f + \sum_{i=1}^{n}(j_i - b) \cdot p_i)$.

EXPRESS specification:

```
*)
ENTITY standard_table_function
  SUBTYPE OF (linearized_table_function);
  order : ordering_type;
WHERE
  WR1: extremal_position_check(SELF);
END_ENTITY;
(*
```

Attribute definitions:

order: Indicator for whether the linear ordering of the multi-dimensional array is the natural ordering (**by_rows**) or the FORTRAN ordering (**by_columns**).

Formal propositions:

WR1: The lowest and highest linear positions that can be computed by the indexing algorithm for valid subscript tuples shall lie in the domain of **SELF\linearized_table_function.source**.

EXAMPLE 1 The $3 \times 4 \times 2$ array whose first and second planes are:

$$\left[\begin{array}{cccc|cccc} 111 & 121 & 131 & 141 & 112 & 122 & 132 & 142 \\ 211 & 221 & 231 & 241 & 212 & 222 & 232 & 242 \\ 311 & 321 & 331 & 341 & 312 & 322 & 332 & 342 \end{array}\right]$$

and whose indices all start from one, can be represented by

```
LOCAL
  tbl342 : standard_table_function := make_standard_table_function(1, [3,4,2],
    rawdata, 1, by_columns);
END_LOCAL;
```

The instance *rawdata* is constructed in the example in 4.5.46. The first argument to **make_standard_table_function** supplies the **index_base** attribute inherited from **explicit_table_function**. The second argument supplies the **shape** attribute inherited from **explicit_table_function**. The third argument

© ISO 2002 — All rights reserved

supplies the **operand** attribute inherited from **unary_generic_expression** but which is also redefined in **linearized_table_function** and, additionally, derived as the inherited **source** attribute. The fourth argument supplies the **first** attribute inherited from **linearized_table_function**. Finally, the fifth argument supplies the **order** attribute of **standard_table_function**.

EXAMPLE 2 The 4×3 matrix

$$\begin{bmatrix} 112 & 212 & 312 \\ 122 & 222 & 322 \\ 132 & 232 & 332 \\ 142 & 242 & 342 \end{bmatrix}$$

which is the transpose of the second plane of the previous table may be represented by an instance of **standard_table_function** as follows:

```
LOCAL
  tbl43 : standard_table_function := make_standard_table_function(1, [4,3],
    rawdata, 13, by_rows);
END_LOCAL;
```

EXAMPLE 3 The $3 \times 4 \times 2$ array whose data is the same as that in Example 1, but whose index ranges are [-1:1], [1:4], and [13:14] can be represented by reindexing the standard table function as follows:

```
LOCAL
  ary342 : reindexed_array_function := make_reindexed_array_function(tbl342,
    [-1,1,13]);
END_LOCAL;
```

4.5.54 regular_table_function

This type of **linearized_table_function** represents dense arrays of any dimension in which the difference in the linear positions corresponding to two subscript tuples can be computed as the inner product of the vector difference of the tuples with a constant tuple.

The value associated with an input subscript tuple $[j_1, j_2, \ldots, j_n]$ is computed as follows: Let L be the function represented by the value of **SELF\linearized_table_function.source**, f be the value of **SELF\linearized_table_function.first**, p_i be the members of **increments**, and b be the value of **SELF\explicit_table_function.index_base**. The table value for $[j_1, j_2, \ldots, j_n]$ is $L(f + \sum_{i=1}^{n}(j_i - b) \cdot p_i)$.

NOTE 1 All slices and subarrays of a standard table function are representable as regular table functions using the same data array (that is, the same **source** attribute value).

NOTE 2 Both standard orderings are special cases of regular order.

NOTE 3 A table obtained by permuting the subscripts of a table in regular order may be expressed as a table in regular order using the same **SELF\linearized_table_function.source** attribute value (that is, without repeating and reordering the list of entries).

NOTE 4 A table obtained by fixing some of the subscript coordinates and/or shrinking the subscript intervals for some of the subscript coordinates of a table in regular order may be expressed as a table in regular order using the same **SELF\linearized_table_function.source** attribute value (i.e without selecting the designated entries and forming a new list or array).

NOTE 5 A table obtained by inverting the order of some of the subscript coordinates may be expressed as a table in regular order using the same (inherited) **source** attribute value.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
ENTITY regular_table_function
  SUBTYPE OF (linearized_table_function);
  increments : LIST [1:?] OF INTEGER;
WHERE
  WR1: SIZEOF (increments) = SIZEOF (self\explicit_table_function.shape);
  WR2: extremal_position_check(self);
END_ENTITY;
(*
```

Attribute definitions:

increments: The constant tuple that expresses the delta in linear position for a unit increment in the corresponding subscript coordinate. May include negative or zero values.

Formal propositions:

WR1: The size of the **increments** tuple shall be the same as the subscript space dimension.

WR2: The lowest and highest linear positions that can be computed by **regular_indexing** for valid subscript tuples shall lie in the domain of **SELF\linearized_table_function.source**.

EXAMPLE 1 The $3 \times 2 \times 2$ array obtained by taking the middle columns of the first example under **standard_table_function**:

$$\begin{bmatrix} 121 & 131 & | & 122 & 132 \\ 221 & 231 & | & 222 & 232 \\ 321 & 331 & | & 322 & 332 \end{bmatrix}$$

and whose indices all start from one, can be represented by

```
LOCAL
  tbl322 : regular_table_function := make_regular_table_function(1, [3,2,2],
    rawdata, 4, [1,3,12]);
END_LOCAL;
```

The instance *rawdata* is constructed in the example in 4.5.46. The critical regularity in the locations of the table entries in instance "rawdata" is that in changing the first index by one, the location changes by one, in changing the second index by one, the location changes by three, and, in changing the third index by one, the location changes by twelve. This regularity is captured by the value supplied for the **increments** attribute.

EXAMPLE 2 The 2×2 submatrix of the preceding table obtained by fixing the first index at the value three is:

$$\begin{bmatrix} 321 & 331 \\ 322 & 332 \end{bmatrix}$$

and it can be represented by

```
LOCAL
  tbl22 : regular_table_function := make_regular_table_function(1, [2,2],
    rawdata, 6, [3,12]);
END_LOCAL;
```

EXAMPLE 3 The $2 \times 3 \times 2$ table obtained by permuting the first two indices of the table in Example

© ISO 2002 — All rights reserved

1 is:

$$\begin{bmatrix} 121 & 221 & 321 & | & 122 & 222 & 322 \\ 131 & 231 & 331 & | & 132 & 232 & 332 \end{bmatrix}$$

and it can be represented by

```
LOCAL
  tbl232 : regular_table_function := make_regular_table_function(1, [2,3,2],
    rawdata, 4, [3,1,12]);
END_LOCAL;
```

4.5.55 triangular_matrix

This type of **linearized_table_function** represents triangular matrices in a compact manner.

EXPRESS specification:

```
*)
ENTITY triangular_matrix
  SUBTYPE OF (linearized_table_function);
  default_entry : maths_value;
  lo_up         : lower_upper;
  order         : ordering_type;
WHERE
  WR1: SIZEOF (SELF\explicit_table_function.shape) = 2;
  WR2: member_of(default_entry, SELF\maths_function.range);
END_ENTITY;
(*
```

Attribute definitions:

default_entry: The special output value used for positions not in the selected lower or upper triangle.

lo_up: Indicator for whether the lower or the upper triangle contains the non-default values.

order: Indicator for whether the linear order of the positions in the selected triangle is by rows or by columns.

Formal propositions:

WR1: The table is two-dimensional.

WR2: The default entry is a member of the function range.

4.5.56 strict_triangular_matrix

This type of **triangular_matrix** represents triangular matrices with a constant main diagonal, which may be the same or different from the constant value **SELF\triangular_matrix.default_entry** used in the opposite triangle.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
ENTITY strict_triangular_matrix
  SUBTYPE OF (triangular_matrix);
  main_diagonal_value : maths_value;
END_ENTITY;
(*
```

Attribute definitions:

main_diagonal_value: The constant value for all entries on the main diagonal.

EXAMPLE The matrix

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.7 & 1.0 & 0.0 \\ 0.3 & 0.5 & 1.0 \end{bmatrix}$$

could be constructed in EXPRESS as:

```
LOCAL
  rdata : listed_real_data := make_listed_real_data(0, [0.7,0.3,0.5]);
  matrix : strict_triangular_matrix := make_strict_triangular_matrix
    (1, [3,3], rdata, 0, 0.0, lower, by_rows, 1.0);
END_LOCAL;
```

The arguments to **make_strict_triangular_matrix** correspond to the explicit attributes as follows: **index_base** and **shape** in **explicit_table_function**, **operand** in **unary_generic_expression**, **first** in **linearized_table_function**, **default_entry**, **lo_up**, and **ordering** in **triangular_matrix**, and **main_diagonal_value** in **strict_triangular_matrix**.

4.5.57 symmetric_matrix

This type of **linearized_table_function** represents several kinds of matrices possessing symmetry about the main diagonal based on an involution of the underlying element space. A matrix is symmetric if the entry for position $[j,k]$ is always the involute of the entry for position $[k,j]$ and vice versa. In the simplest case, the involution is the identity mapping and the entries in corresponding positions are the same. The other cases represented are skew-symmetry based on negation of numbers as the involution, Hermitian symmetry based on conjugation of complex numbers as the involution, and skew-Hermitian symmetry based on the negative conjugate involution of the complex numbers.

The value of **triangle** indicates which “half” of the matrix is represented directly by the values from **SELF\linearized_table_function.source**. The other “half” of the matrix is computed from the indicated half by using the involution. In the case of a skew-symmetric matrix, the main diagonal is necessarily zero, so the indicated triangle is the strict one not including that diagonal.

The value of **order** indicates the order in which values are indexed in **SELF\linearized_table_function.source**

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
ENTITY symmetric_matrix
  SUBTYPE OF (linearized_table_function);
  symmetry : symmetry_type;
  triangle : lower_upper;
  order    : ordering_type;
WHERE
  WR1: SIZEOF (SELF\explicit_table_function.shape) = 2;
  WR2: SELF\explicit_table_function.shape[1] =
       SELF\explicit_table_function.shape[2];
  WR3: NOT (symmetry = skew) OR (
       (space_dimension(SELF\linearized_table_function.source.range) = 1) AND
        subspace_of_es(factor1(SELF\linearized_table_function.source.range),
        es_numbers));
  WR4: NOT ((symmetry = hermitian) OR (symmetry = skew_hermitian)) OR (
       (space_dimension(SELF\linearized_table_function.source.range) = 1) AND
        subspace_of_es(factor1(SELF\linearized_table_function.source.range),
        es_complex_numbers));
END_ENTITY;
(*
```

Attribute definitions:

symmetry: Indicates the kind of symmetry present in the matrix.

triangle: Indicates whether the values for the upper triangle or the lower triangle of the matrix are provided by the inherited **SELF\linearized_table_function.source** attribute.

order: Indicator for whether the values from **SELF\linearized_table_function.source** are ordered by rows or by columns.

Formal propositions:

WR1: The table is two-dimensional.

WR2: The number of rows shall be equal to the number of columns.

WR3: If the matrix is skew-symmetric, the output shall be single numbers.

WR4: If the matrix is Hermitian or skew-Hermitian, the output shall be single complex numbers.

4.5.58 symmetric_banded_matrix

This type of **symmetric_matrix** represents symmetric banded matrices. A symmetric banded matrix is one which is both symmetric and banded.

NOTE Subtyping only from **symmetric_matrix** and not **banded_matrix** avoids extraneous attributes (**below**) and duplication of attributes (**order**).

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
ENTITY symmetric_banded_matrix
  SUBTYPE OF (symmetric_matrix);
  default_entry : maths_value;
  above         : nonnegative_integer;
WHERE
  WR1: member_of(default_entry,
       factor1(SELF\linearized_table_function.source.range));
END_ENTITY;
(*
```

Attribute definitions:

default_entry: The constant output value for all positions $[j, k]$ for which $|k - j| >$ **above** in the indicated upper or lower triangle of the symmetric matrix, that is, positions outside the band in the indicated triangle. The output value for the positions positions outside the band in the other triangle is determined by the kind of symmetry indicated by **symmetry**.

above: The number of diagonals above or below the main diagonal which may contain non-default entries supplied by **SELF\linearized_table_function.source**.

Formal propositions:

WR1: The default value is a member of the range of the inherited **source** function.

4.5.59 banded_matrix

This type of **linearized_table_function** represents banded matrices. A banded matrix is one in which the non-default values all lie in a relatively small number of consecutive diagonals.

EXPRESS specification:

```
*)
ENTITY banded_matrix
  SUBTYPE OF (linearized_table_function);
  default_entry : maths_value;
  below         : integer;
  above         : integer;
  order         : ordering_type;
WHERE
  WR1: SIZEOF (self\explicit_table_function.shape) = 2;
  WR2: -below <= above;
  WR3: member_of(default_entry,
       factor1(SELF\linearized_table_function.source.range));
END_ENTITY;
(*
```

© ISO 2002 — All rights reserved

Attribute definitions:

default_entry: The constant output value for all positions $[j, k]$ for which $j - k >$ **below** or $k - j >$ **above**, that is, all positions outside the band.

below: The number of diagonals below the main diagonal which may contain non-default entries.

above: The number of diagonals above the main diagonal which may contain non-default entries.

order: Indicator for whether the values from **SELF\linearized_table_function.source** are ordered by rows or by columns.

Formal propositions:

WR1: The table is two-dimensional.

WR2: At least one diagonal shall have non-default entries.

WR3: The default value is a member of the range of the inherited **source** function.

NOTE Negative values are allowed for the attributes **below** and **above**. A banded matrix in which all the non-default entries are in the first diagonal above the main diagonal can be represented efficiently using the value -1 for **below** and +1 for **above**.

EXAMPLE Consider the 1000×1000 real matrix indexed from zero with 2^{-n} on its main diagonal, the constant minus one on the diagonal above the main diagonal, and zeroes elsewhere. To represent this matrix using an instance of **banded_matrix**, one must first represent the non-default values in **by_row** or **by_column** order. The values in **by_rows** order are

$$2^0, -1, 2^{-1}, -1, 2^{-2}, -1, 2^{-3}, -1, ...$$

Assuming the sequence is indexed by n starting from zero, the function expressing this sequence is

$$f(n) \equiv \left\{ \begin{array}{ll} 2^{-n/2}, & \text{if } n \bmod 2 = 0 \\ -1, & \text{otherwise} \end{array} \right\}.$$

Such a function may be represented by a tree of instances of **parallel_composed_function** as follows:

```
LOCAL
  intsge0 : integer_interval_from_min := make_integer_interval_from_min( 0);
  sel1 : selector_function := make_selector_function( 1, intsge0);
  coni2 : constant_function := make_constant_function( 2, intsge0);
  nmod2 : parallel_composed_function := make_parallel_composed_function(
    intsge0, [sel1, coni2], ef_mod_i);
  coni0 : constant_function := make_constant_function( 0, intsge0);
  nmod2eq0 : parallel_composed_function := make_parallel_composed_function(
    intsge0, [nmod2, coni0], ef_eq_i);
  nover2 : parallel_composed_function := make_parallel_composed_function(
    intsge0, [sel1, coni2], ef_divide_i);
  negnover2 : parallel_composed_function := make_parallel_composed_function(
    intsge0, [nover2], ef_negate_i);
  conr2 : constant_function := make_constant_function( 2.0, intsge0);
  powerof2 : parallel_composed_function := make_parallel_composed_function(
    intsge0, [conr2, negnover2], ef_exponentiate_ri);
  conrneg1 : constant_function := make_constant_function( -1.0, intsge0);
  fn : parallel_composed_function := make_parallel_composed_function(
    intsge0, [nmod2eq0, powerof2, conrneg1], ef_if_r);
```

 © ISO 2002 — All rights reserved

```
  bandmat : banded_matrix := make_banded_matrix( 0, [1000,1000], fn, 0, 0.0,
    0, 1, by_rows);
END_LOCAL;
```

Observe that this matrix has one million entries, 1999 of them non-zero, but its structure is sufficiently simple that it can be represented by just thirteen instances in twenty lines.

4.5.60 basic_sparse_matrix

This type of **explicit_table_function** and **multiple_arity_generic_expression** represents sparse matrices. A sparse matrix is one in which most entries are a single constant value. The representation lists the non-defaulted positions and their corresponding entries, ordered and indexed in a manner which supports efficient searching.

To evaluate the sparse matrix for the position $[j, k]$ when **order** is **by_rows**, 1) evaluate **index** at position j to obtain mlo, 2) evaluate **index** at position $(j+1)$ to obtain mhi, 3) search **loc** at positions $mlo <= m < mhi$ looking for the entry k, 4) if evaluation of **loc** at position m returns k, then evaluate **val** at position m to obtain the sparse matrix entry for position $[j, k]$, otherwise, 5) (k not an entry in **loc** in the computed interval), the sparse matrix entry for position $[j, k]$ is **default_entry**. If the **order** is **by_columns**, the roles of j and k are reversed.

The input to the function represented by a **basic_sparse_matrix** is a single ordered pair of integers and the output is a single value from the range of **val**.

NOTE Other, more specialized representations for sparse matrices may be required for certain applications in the future. They may take advantage of additional structure that may exist in the matrices used by these applications.

EXPRESS specification:

```
*)
ENTITY basic_sparse_matrix
  SUBTYPE OF (explicit_table_function, multiple_arity_generic_expression);
  SELF\multiple_arity_generic_expression.operands : LIST [3:3] OF maths_function;
  default_entry : maths_value;
  order : ordering_type;
DERIVE
  index : maths_function := SELF\multiple_arity_generic_expression.operands[1];
  loc   : maths_function := SELF\multiple_arity_generic_expression.operands[2];
  val   : maths_function := SELF\multiple_arity_generic_expression.operands[3];
WHERE
  WR1: function_is_1d_table(index);
  WR2: function_is_1d_table(loc);
  WR3: function_is_1d_table(val);
  WR4: check_sparse_index_domain(index.domain, index_base, shape, order);
  WR5: check_sparse_index_to_loc(index.range, loc.domain);
  WR6: loc.domain = val.domain;
  WR7: check_sparse_loc_range(loc.range, index_base, shape, order);
  WR8: member_of(default_entry, val.range);
END_ENTITY;
(*
```

© ISO 2002 — All rights reserved

Attribute definitions:

SELF\multiple_arity_generic_expression.operands: The inherited **operands** attribute is constrained to be a list of three instances of **maths_function**.

default_entry: The constant output value for all positions not specifically identified as having a non-default value.

order: Indicator for whether the non-default output values are ordered by rows or by columns.

index: Function providing starting locations in **loc** and **val** for the non-default pairs in each row (if **order** = **by_rows**) or column (if **order** = **by_columns**). Derived to be the same as the first operand.

loc: Function providing the column indices of the non-default positions (if **order** = **by_rows**) or the row indices of the non-default positions (if **order** = **by_columns**) in the order determined for the non-default positions by **order**. Derived to be the same as the second operand.

val: Function providing the output values for the non-default positions in one-to-one correspondence with **loc**. Derived to be the same as the third operand.

Formal propositions:

WR1: The value of **index** is a one-dimensional table function.

WR2: The value of **loc** is a one-dimensional table function.

WR3: The value of **val** is a one-dimensional table function.

WR4: If **order** is **by_rows**, then the domain of **index** is the space of row positions of the matrix plus an extra row, and if **order** is **by_columns**, then the domain of **index** is the space of column positions of the matrix plus an extra column.

WR5: The outputs from **index** shall be members of the domain of the **loc** function extended by one integer.

WR6: The domain of the **loc** function shall be the same as the domain of the **val** function.

WR7: If **order** is **by_rows** then the outputs from **loc** shall be column positions of the matrix, and if **order** is **by_columns** then the outputs from **loc** shall be row positions.

WR8: The default value shall be a member of the range of **val**.

EXAMPLE The matrix

$$\begin{bmatrix} 0.0 & 2.1 & 3.6 & 0.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -7.1 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -8.5 & 2.3 & 1.0 \\ 0.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

could be represented as an instance of **basic_sparse_matrix** via

```
LOCAL
  index : listed_integer_data := make_listed_integer_data(1, [1,3,4,5,8,9]);
  loc : listed_integer_data := make_listed_integer_data(1, [2,3,1,3,4,5,6,2]);
  val : listed_real_data := make_listed_real_data(1,
    [2.1,3.6,5.0,-7.1,-8.5,2.3,1.0,2.0]);
  sparse : basic_sparse_matrix := make_basic_sparse_matrix(1, [5,6],
    [index, loc, val], 0.0, by_rows);
END_LOCAL;
```

 © ISO 2002 — All rights reserved

The "val" instance lists the non-defaulted entries in the order encountered when traversing the matrix by rows. The "loc" instance lists the column locations of the non-defaulted entries. The "index" instance lists the locations in the "val" and "loc" lists of the first non-defaulted entry in each row. The arguments to **make_basic_sparse_matrix** correspond to the explicit attributes as follows: **index_base** and **shape** in **explicit_table_function**, **operands** in **multiple_arity_generic_expression** which is redefined in **basic_sparse_matrix**, and **default_entry** and **order** in **basic_sparse_matrix**.

4.5.61 homogeneous_linear_function

This type of **maths_function** and **unary_generic_expression** provides compact representations for multidimensional homogeneous linear functions. Such mathematical functions preserve the vector space operations of vector addition and scalar multiplication, here applied to number tuples without prejudice as to what they might or might not represent. Such functions are completely defined by a matrix and a rule for how the matrix and the input tuple are to be "matrix-multiplied". The matrix multiplication to be used is indicated by identifying which of the two matrix indices is to be summed over. When the first index is the summation index, the effect is that of multiplying the input tuple considered as a row vector by the matrix on the right. When the second index is the summation index, the effect is that of multiplying the input tuple considered as a column vector by the matrix on the left.

The functions represented by this type are conceived as taking a single tuple as input and producing a single tuple as output.

NOTE 1 The indication of summation indices generalizes more easily to other contexts.

NOTE 2 The matrix of coefficients may be given by any two-dimensional table function. In particular, it might be an instance of **standard_table_function**, **regular_table_function**, **triangular_matrix**, **sparse_matrix**, **parallel_composed_function**, or many other types. The critical characteristics identifying a two-dimensional table function have to do with its domain and range and not with how its algorithm is specified.

EXPRESS specification:

```
*)
ENTITY homogeneous_linear_function
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
  sum_index : one_or_two;
DERIVE
  mat       : maths_function := SELF\unary_generic_expression.operand;
WHERE
  WR1: function_is_2d_table(mat);
  WR2: (space_dimension(mat.range) = 1) AND
       subspace_of_es(factor1(mat.range),es_numbers);
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be a **maths_function**.

sum_index: The ordinal indicating which subscript index of the matrix appears as the sum-

© ISO 2002 — All rights reserved

mation index.

mat: The function giving the matrix of coefficients of the homogeneous linear function. Derived to be the same as the inherited **operand** value.

Formal propositions:

WR1: : The function providing the matrix of coefficients is a two-dimensional table function.

WR2: The range space of **mat** shall be one-tuples of a subspace of the space of all numbers. In other words, the matrix shall be numeric.

4.5.62 general_linear_function

This type of **maths_function** and **unary_generic_expression** provides compact representations for multidimensional general linear functions. Like the homogeneous linear function, such a function is completely determined by a matrix of coefficients, in this case including an extra column or row (depending on the summation index) for the constant terms. Conceptually, the representation is accomplished by composing the canonical mapping from ordinary coordinates to homogeneous coordinates with a homogeneous linear mapping in the space of homogeneous coordinates.

In other words, the "translation vector" is included in the matrix as an extra row if using **sum_index** = 1 or an extra column if using **sum_index** = 2, and the input tuple has a one appended to it before the "matrix multiplication" is carried out.

The functions represented by this type are conceived as taking a single tuple as input and producing a single tuple as output.

NOTE 1 General linear functions include all the "rigid motions" such as translation, rotation, and reflection, as well as "nonrigid motions" such as rescaling, projecting and shearing.

NOTE 2 General linear functions are precisely those functions which preserve affine subspaces of a real space, that is, they map lines to lines, planes to planes, and n-dimensional "flat" subspaces to n-dimensional "flat' subspaces for any positive integer n.

EXPRESS specification:

```
*)
ENTITY general_linear_function
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
  sum_index : one_or_two;
DERIVE
  mat       : maths_function := SELF\unary_generic_expression.operand;
WHERE
  WR1: function_is_2d_table(mat);
  WR2: (space_dimension(mat.range) = 1) AND
       subspace_of_es(factor1(mat.range),es_numbers);
END_ENTITY;
(*
```

 © ISO 2002 — All rights reserved

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be a **maths_function**.

sum_index: The subscript index of the matrix which appears in the summation.

mat: The function giving the matrix of coefficients of the general linear function. Derived to be the same as the inherited **operand** value.

Formal propositions:

WR1: : The function providing the matrix of coefficients is a two-dimensional table function.

WR2: The range space of **mat** shall be one-tuples of a subspace of the space of all numbers. In other words, the matrix shall be numeric.

EXAMPLE The general linear function $\mathbf{F}(\mathbf{x}) \equiv \mathbf{A}\mathbf{x} + \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 0.0 & 2.1 & 3.6 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -7.1 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -8.5 & 2.3 \\ 0.0 & 2.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 1.0 \\ 0.0 \end{bmatrix}$$

could be represented as an instance of **general_linear_function** via

```
LOCAL
  axplusb : general_linear_function := make_general_linear_function(sparse, 2);
END_LOCAL;
```

Observe that the matrix described in the example of **basic_sparse_matrix** is matrix **A** with **b** adjoined as an additional column.

4.5.63 b_spline_basis

This type of **maths_function** and **generic_literal** defines a tuple of B-spline basis functions, usually for use with one of the input variables of a B-spline function.

The function represented by an instance of this type takes a single real number as input and produces a real tuple of length **num_basis** as output.

EXPRESS specification:

```
*)
ENTITY b_spline_basis
  SUBTYPE OF (maths_function, generic_literal);
  degree         : nonnegative_integer;
  repeated_knots : LIST [2:?] OF REAL;
DERIVE
  order          : positive_integer := degree + 1;
  num_basis      : positive_integer := SIZEOF (repeated_knots) - order;
WHERE
  WR1: num_basis >= order;
  WR2: nondecreasing(repeated_knots);
```

© ISO 2002 — All rights reserved

```
  WR3: repeated_knots[order] < repeated_knots[num_basis+1];
END_ENTITY;
(*
```

Attribute definitions:

degree: The degree of the basis functions as piecewise polynomials.

repeated_knots: The knot sequence which determines the basis functions.

order: The order of the B-spline representation. Equivalently, the number of degrees of freedom in the polynomials making up the basis functions. Equivalently, one more than the degree of the polynomials making up the basis functions.

num_basis: The number of basis functions.

Formal propositions:

WR1: The computed number of basis functions in a B-spline basis shall be greater than or equal to the order.

WR2: The knots shall be arranged in non-decreasing order.

WR3: The parametric domain is an interval of positive length.

NOTE 1 **order** is more directly useful than **degree** in most computations. Similarly, the knot sequence is what is directly used by the evaluation algorithms, rather than the breakpoints and multiplicities.

NOTE 2 Note that no bound is placed on the number of times a knot is repeated. This makes the communication of general B-spline functions robust under truncation of precision. However, it requires more care than is customary to properly handle such “B-splines with coalesced knots”.

4.5.64 b_spline_function

This type of **maths_function** and **unary_generic_expression** represents tensor product B-spline functions with arbitrary numbers of input and output variables. Depending on the dimension of the coefficient table, the outputs are real numbers, real tuples, or multi-dimensional real table functions.

EXPRESS specification:

```
*)
ENTITY b_spline_function
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
  basis : LIST [1:?] OF b_spline_basis;
DERIVE
  coef  : maths_function := SELF\unary_generic_expression.operand;
WHERE
  WR1: function_is_table(coef);
  WR2: (space_dimension(coef.range) = 1) AND
       (number_superspace_of(factor1(coef.range)) = the_reals);
  WR3: SIZEOF (basis) <=
       SIZEOF (shape_of_array(coef));
```


© ISO 2002 — All rights reserved

```
  WR4: compare_basis_and_coef(basis, coef);
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be a **maths_function**.

basis: The list of entities defining the B-spline basis functions for each input variable.

coef: The multi-dimensional table function providing the coefficients for each combination of a tensor product of basis functions and an elementary output variable. Derived to be the same as the inherited **operand** attribute.

Formal propositions:

WR1: The function providing the coefficients is a table function.

WR2: The coefficient array function is real-valued.

WR3: The size of the **basis** list is less than or equal to the dimension of the coefficient table.

WR4: The numbers of basis functions in the list of B-spline bases match one-for-one with the initial values of the coefficient table shape tuple.

EXAMPLE The following constructs an instance of **b_spline_function** which describes a piecewise quadratic curve in a four dimensional mathematical space.

```
LOCAL
  basis : b_spline_basis := make_b_spline_basis(2, [0.00,0.00,0.00,0.25,0.25,
    0.50,0.50,0.75,0.75,1.00,1.00,1.00]);
  rt2 : REAL := SQRT(2.0);
  rrt2 : REAL := 1.0 / SQRT(2.0);
  rawcoef : listed_real_data := make_listed_real_data(0,
    [  1.0, rrt2,  0.0,-rrt2, -1.0,-rrt2,  0.0, rrt2,  1.0,
       0.0, rrt2,  1.0, rrt2,  0.0,-rrt2, -1.0,-rrt2,  0.0,
       2.0,  rt2,  2.0,  rt2,  2.0,  rt2,  2.0,  rt2,  2.0,
       1.0, rrt2,  1.0, rrt2,  1.0, rrt2,  1.0, rrt2,  1.0]);
  coef : standard_table_function := make_standard_table_function(0, [9,4],
    rawcoef, 0, by_columns);
  crv4d : b_spline_function := make_b_spline_function(coef, [basis]);
END_LOCAL;
```

4.5.65 rationalize_function

This type of **maths_function** and **unary_generic_expression** represents mathematical functions resulting from a composition of a number-tuple producing function with a canonical mapping from homogeneous coordinates to ordinary coordinates. The canonical mapping creates output tuples by taking the ratios of all but the last component of the input tuple with the last component. Where a last component has a zero value, the **rationalize_function** is undefined.

The functions represented by this type take a single number tuple as input and produce a single number tuple as output. The type of the numbers is derived to be the same as the type of the numbers produced by **fun**.

© ISO 2002 — All rights reserved

NOTE The main inspiration for this type is to provide a compact representation for rational B-spline functions. Most B-spline functions produced by applications are not rational, and the algorithms for working with such non-rational B-spline functions are much simpler than for rational functions. On the other hand, the operation which converts a non-rational function in homogeneous coordinates to a rational function in ordinary coordinates is applicable to any kind of function. Thus, the separation of the non-rational B-splines from the rational makes possible both a simplification and also a generalization.

Rational B-spline functions can be represented by instantiating a **rationalize_function** with a **fun** attribute value set to a **b_spline_function** which corresponds very simply with the desired rational B-spline function. Numerator coefficients X_{ij}, Y_{ij}, Z_{ij} and denominator (or weights) coefficients W_{ij}, become spline coefficients $X_{ij}W_{ij}, Y_{ij}W_{ij}, Z_{ij}W_{ij}$, and W_{ij} in a non-rational spline.

EXPRESS specification:

```
*)
ENTITY rationalize_function
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
DERIVE
  fun : maths_function := SELF\unary_generic_expression.operand;
WHERE
  WR1: (space_dimension(fun.domain) = 1) AND (space_dimension(fun.range) = 1);
  WR2: number_tuple_subspace_check(factor1(fun.range));
  WR3: space_dimension(factor1(fun.range)) > 1;
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be a **maths_function**.

fun: The function whose output tuples are treated as homogeneous coordinates and from which the rationalized function is constructed. Derived to be the same as the inherited **operand** attribute.

Formal propositions:

WR1: The value of attribute **fun** is a function taking one input and producing one output.

WR2: The outputs of **fun** shall be number tuples.

WR3: The size of the tuples output by **fun** shall be greater than one.

EXAMPLE The four-dimensional B-spline curve represented by the instance "crv4d" of **b_spline_function** in the preceding example can be used to form a rational B-spline representation of the unit circle in the plane $z = 2$ using an instance of **rationalize_function** as follows:

```
LOCAL
  circ : rationalize_function := make_rationalize_function(crv4d);
END_LOCAL;
```

This usage causes the last coordinate of "crv4d" to be used as the weights or denominator of the rational function.

 © ISO 2002 — All rights reserved

4.5.66 partial_derivative_function

This type of **maths_function** and **unary_generic_expression** represents a specified partial derviative of another function. Which partial derivative is represented shall be specified by indicating the sequence of partial derivative operators using values of type **input_selector** corresponding to the implicit input variables. If the domain is one-tuples of a tuple space, the implicit input variables correspond to the components of that tuple space. Otherwise, the implicit input variables correspond to the components of the domain. See /crefinp:sel.

The domain of the **partial_derivative_function** is derived to be the same as that of the function being differentiated.

The range of the **partial_derviative_function** is derived from that of the function being differentiated by removing any restrictions to subsets of the real or complex numbers, but retaining the tuple structure and number types of the outputs, tuple components, tuple subcomponents, and so forth.

The extension of the exact domain of a partial derivative function to include boundary and other problematic inputs shall be specified by the value of the attribute **extension**.

NOTE 1 The strictest definition of derivative requires that the function being differentiated be defined on an open set containing the point at which the derivative is being evaluated. This implies that the derivative is never defined at boundaries of the domain. More commonly, the derivative is extended to all points at which continuity of the derivative can be preserved. In the case of the real numbers, one can also define left- and right-sided derivatives which may be defined at places the strict (two-sided) derivative is not. The **extension** attribute requires specification of which of these common extensions to the strictest notion of derivative is to be represented.

NOTE 2 Partial derivatives only exist mathematically for certain functions. At a minimum, the outputs must be real or complex numbers or (possibly nested) tuples thereof and the inputs corresponding to members of **d_variables** must be real or complex numbers. The **space_is_continuum** and **partial_-derivative_check** functions verify these minimal requirements.

EXPRESS specification:

```
*)
ENTITY partial_derivative_function
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
  d_variables : LIST [1:?] OF input_selector;
  extension : extension_options;
DERIVE
  derivand : maths_function := SELF\unary_generic_expression.operand;
WHERE
  WR1: space_is_continuum (derivand.range);
  WR2: partial_derivative_check (derivand.domain, d_variables);
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be a **maths_function**.

© ISO 2002 — All rights reserved

d_variables: The list of partial derivative operations to be applied to **derivand**, indicated by values of type **input_selector** corresponding to implicit input variables. The order of the partial differentiation operations is the order implied by the list.

extension: Enumeration value indicating the policy to be followed in evaluating this instance at boundary and other problematic inputs.

derivand: The function whose partial derivative is to be represented. Derived to be the same as the inherited **operand** attribute.

Formal propositions:

WR1: The ultimate components of the outputs of **derivand** shall be real or complex numbers and any packaging into nested tuples shall have a fixed structure. Equivalently, the range of **derivand** shall be a cartesian product of subspaces of the reals or the complex numbers or cartesian products of such subspaces nested to arbitrary finite depths.

WR2: The members of **d_variables** shall correspond to implicit input variables of **derivand** and the inputs corresponding to them shall be real or complex numbers.

EXAMPLE Represent the second partial derivative of the function $f(x,y) \equiv xy^2$, differentiated first with respect to y and then x. That is, represent

$$\frac{\partial^2 f}{\partial x \partial y}.$$

```
LOCAL
  r2 : uniform_product_space := make_uniform_product_space(the_reals,2);
  sel1 : selector_function := make_selector_function(1,r2);
  sel2 : selector_function := make_selector_function(2,r2);
  f : parallel_composed_function := make_parallel_composed_function(r2,
    [sel1,sel2,sel2], ef_multiply_r);
  d2f_dxdy : partial_derivative_function := make_partial_derivative_function(
    f, [2,1], eo_cont);
END_LOCAL;
```

Technically, the preceding example represented $f(x,y) \equiv xyy$. It would take two additional entity instances to represent xy^2 exactly as written. See also essentially the same example below, but represented using a **partial_derivative_expression** instance.

4.5.67 partial_derivative_expression

This type of **unary_generic_expression** represents a specified partial derviative of another expression. Which partial derivative is represented shall be specified by indicating the list of variables for which the process of partial differentiation is to be carried out.

The extension of the exact domain of a partial differentiation process to accept boundary and other problematic value assignments to relevant variables shall be specified by the value of the attribute **extension**.

EXPRESS specification:

```
*)
ENTITY partial_derivative_expression
```

 © ISO 2002 — All rights reserved

```
  SUBTYPE OF (unary_generic_expression);
  d_variables : LIST [1:?] OF maths_variable;
  extension : extension_options;
DERIVE
  derivand : generic_expression := SELF\unary_generic_expression.operand;
WHERE
  WR1: has_values_space (derivand);
  WR2: space_is_continuum (values_space_of (derivand));
  WR3: SIZEOF (QUERY (vbl <* d_variables | (NOT subspace_of (values_space_of (vbl),
    the_reals)) AND (NOT subspace_of (values_space_of (vbl), the_complex_numbers))
    )) = 0;
END_ENTITY;
(*
```

Attribute definitions:

d_variables: The list of partial differentiation variables. The order of the partial differentiation processes is the order implied by the list.

extension: Enumeration value indicating the policy to be followed in evaluating this expression when boundary or other problematic inputs are assigned to the variables.

derivand: The expression to be differentiated. Derived to be the inherited **SELF\unary_-generic_expression.operand**.

Formal propositions:

WR1: The expression to be differentiated shall have a values space.

WR2: The values space of the expression to be differentiated shall be a continuum, that is, consist of real or complex numbers or nested tuples thereof in a static structure.

WR3: The differentiation variables shall range over real or complex numbers.

NOTE 1 Although **partial_derivative_expression** is theoretically redundant after defining **partial_-derivative_function**, the formation of an equivalent expression using **abstracted_expression_function**, **partial_derivative_function**, **function_application**, and **expression_denoted_function** is too complex to be practical. The situation is similar in using **partial_derivative_expression** to define **partial_derivative_function**. Hence, both entity types are defined.

NOTE 2 The common notations for the differentiation process conceal some complex and subtle matters. Of special concern here is the role of the differentiation variables in the expression. Upon careful examination, it will be found that each differentiation variable becomes bound (by lambda abstraction), two new corresponding variables are introduced (in forming the divided difference expression) and one of those is bound again (in taking the limit as it goes to zero). Consequently, the **partial_derivative_expression** does not really contain the same variables as the expression being differentiated, it contains new variables in one-to-one correspondence to those variables and customarily given the same names. This situation has serious implications for any system computing with these expressions, but can be ignored in the static environment of data exchange. Notice that this issue does not arise for **partial_derivative_function** because there are no objects explicitly representing variables and any implicit variables are entirely local to individual function objects.

EXAMPLE Represent the second partial derivative of the expression xyy, differentiated first with respect to y and then x. That is, represent

$$\frac{\partial^2}{\partial x \partial y}(xyy).$$

© ISO 2002 — All rights reserved

```
LOCAL
  sem : free_variable_semantics := free_variable_semantics() || variable_semantics();
  var_x : maths_real_variable := make_maths_real_variable(the_reals, 'x');
  env_x : environment := make_environment(var_x, sem);
  var_y : maths_real_variable := make_maths_real_variable(the_reals, 'y');
  env_y : environment := make_environment(var_y, sem);
  xyy : mult_expression := make_mult_expression([var_x, var_y, var_y]);
  d2_dxdy_xyy : partial_derivative_expression := make_partial_derivative_expression(
    xyy, [var_y,var_x], eo_cont);
END_LOCAL;
```

See also this same example represented using a **partial_derivative_function**.

4.5.68 definite_integral_function

This type of **maths_function** and **unary_generic_expression** represents functions defined by a definite integral of another function over a real interval. It includes the special cases of intervals of integration unbounded at either or both ends.

The finite bounds of the interval of integration are new initial inputs to the represented function and the remaining inputs are inherited, in the same order, from the integrand function, except that the input corresponding to the variable of integration is omitted.

The represented function has the same number of outputs, of the same real or complex types, and with the same packaging, as the integrand function.

The domains of any new inputs are derived to be the same as the domain of the variable of integration. The range is derived to have the same structure as that of the integrand function, but any restrictions to subsets of the real or complex numbers are replaced by the corresponding spaces of all real numbers and all complex numbers.

EXPRESS specification:

```
*)
ENTITY definite_integral_function
  SUBTYPE OF (maths_function, unary_generic_expression);
  SELF\unary_generic_expression.operand : maths_function;
  variable_of_integration : input_selector;
  lower_limit_neg_infinity : BOOLEAN;
  upper_limit_pos_infinity : BOOLEAN;
DERIVE
  integrand : maths_function := SELF\unary_generic_expression.operand;
WHERE
  WR1: space_is_continuum (integrand.range);
  WR2: definite_integral_check (integrand.domain, variable_of_integration,
    lower_limit_neg_infinity, upper_limit_pos_infinity);
END_ENTITY;
(*
```

Attribute definitions:

SELF\unary_generic_expression.operand: The inherited **operand** attribute is constrained to be a **maths_function**.

 © ISO 2002 — All rights reserved

variable_of_integration: The value of type **input_selector** corresponding to the variable of integration in the implicit list of input variables of the integrand. See 4.4.14.

lower_limit_neg_infinity: Indicator for whether the lower limit of the interval of integration is negative infinity, or a new input.

upper_limit_pos_infinity: Indicator for whether the upper limit of the interval of integration is positive infinity, or a new input.

integrand: The function whose integral is to be represented. Derived to be the same as the inherited **operand** attribute.

Formal propositions:

WR1: The range of the integrand function shall be a continuum, that is, consist of real or complex numbers or nested tuples thereof in a static structure.

WR2: There shall be an input of type real corresponding to the variable of integration, if the lower limit of integration is negative infinity, then the domain of the variable of integration shall be unbounded below, and if the upper limit of integration is positive infinity, then the domain of the variable of integration shall be unbounded above.

EXAMPLE Represent the definite integral of the function $f(x, y) \equiv xyy$ with respect to y from one to any given value z. That is, represent

$$F(z, x) \equiv \int_1^z f(x, y)dy.$$

See the example following **partial_derivative_function** for the construction of the instance representing the function f.

```
LOCAL
  igrlf : definite_integral_function := make_definite_integral_function(
    f, 2, false, false);
END_LOCAL;
```

4.5.69 definite_integral_expression

This type of **quantifier_expression** represents a definite integral of another expression over a real interval. It includes the special cases of intervals of integration unbounded at either or both ends.

The finite bounds of the interval of integration are operands to the integral expression, whenever their existence is indicated by the values of the attributes **lower_limit_neg_infinity** and **upper_limit_pos_infinity**. When both bounds are present, the lower limit is the third operand and the upper limit is the fourth.

EXPRESS specification:

```
*)
ENTITY definite_integral_expression
  SUBTYPE OF (quantifier_expression);
  lower_limit_neg_infinity : BOOLEAN;
  upper_limit_pos_infinity : BOOLEAN;
DERIVE
```

© ISO 2002 — All rights reserved

```
  integrand : generic_expression
    := SELF\multiple_arity_generic_expression.operands[1];
  variable_of_integration : maths_variable
    := SELF\multiple_arity_generic_expression.operands[2];
  SELF\quantifier_expression.variables : LIST [1:1] OF UNIQUE generic_variable
    := [variable_of_integration];
WHERE
  WR1: has_values_space (integrand);
  WR2: space_is_continuum (values_space_of (integrand));
  WR3: definite_integral_expr_check (SELF\multiple_arity_generic_expression.operands,
    lower_limit_neg_infinity, upper_limit_pos_infinity);
END_ENTITY;
(*
```

Attribute definitions:

lower_limit_neg_infinity: Indicator for whether the lower limit of the interval of integration is negative infinity, or an operand.

upper_limit_pos_infinity: Indicator for whether the upper limit of the interval of integration is positive infinity, or an operand.

integrand: The expression to be integrated. Derived to be the first operand in the list inherited from supertype **multiple_arity_generic_expression**.

variable_of_integration: The variable of integration. Derived to be the second operand in the list inherited from supertype **multiple_arity_generic_expression**.

SELF\quantifier_expression.variables: The inherited **variables** attribute is constrained to be the singleton list of the variable of integration.

Formal propositions:

WR1: The expression to be integrated shall have a values space.

WR2: The values space of the expression to be integrated shall be a continuum, that is, consist of real or complex numbers or nested tuples thereof in a static structure.

WR3: The number of operands shall be two plus the number of FALSE values among **lower_limit_neg_infinity** and **upper_limit_pos_infinity**, the values space of the variable of integration shall be a subspace of the real numbers, and the values spaces of the third and fourth operands, when present, shall be compatible with the values space of the variable of integration.

NOTE 1 Although **definite_integral_expression** is theoretically redundant after defining **definite_integral_function**, the formation of an equivalent expression using **abstracted_expression_function**, **definite_integral_function**, **function_application**, and **expression_denoted_function** requires too many instances to be practical. The situation is similar in using **definite_integral_expression** to define **definite_integral_function**. Hence, both entity types are defined.

NOTE 2 Introduction of a new type named **extended_reals** was considered and rejected. Ultimately, it introduced more complexities than simplifications.

EXAMPLE Represent the definite integral of the expression xyy with respect to y from 1 to z. That is, represent

$$\int_1^z xyydy.$$

 © ISO 2002 — All rights reserved

Contrast the example following **partial_derivative_expression** with the example below. Most of the construction must be redone to reflect the change in the semantics of y from free variable to bound variable.

```
LOCAL
  semb : variable_semantics := bound_variable_semantics() || variable_semantics();
  var_yb : maths_real_variable := make_maths_real_variable(the_reals, 'y');
  env_yb : environment := make_environment(var_yb, semb);
  xyyb : mult_expression := make_mult_expression([var_x, var_yb, var_yb]);
  con_1 : real_literal := make_real_literal(1.0);
  var_z : maths_real_variable := make_maths_real_variable(the_reals, 'y');
  env_z : environment := make_environment(var_z, sem);
  igrl_xyy : definite_integral_expression := make_definite_integral_expression(
    [xyyb, var_yb, con_1, var_z], false, false);
END_LOCAL;
```

4.5.70 abstracted_expression_function

This type of **maths_function** and **quantifier_expression** defines a mathematical function by function abstraction from an expression. A list of variables is identified as corresponding to the inputs of the function. These variables become bound variables in the resulting expression, which now denotes a function. That is, the variables identified as function inputs may not be assigned values or any semantics other than that they are placeholders for inputs to a function. This operation corresponds to "lambda quantification" in the Lambda Calculus formalism.

The domain of an instance of **abstracted_expression_function** is derived to be the Cartesian product of the value spaces of its **variables**, in that order. The range is derived to be the value space of its **expr**. The value space of any mathematical expression is computed by function **value_space_of**.

EXPRESS specification:

```
*)
ENTITY abstracted_expression_function
  SUBTYPE OF (maths_function, quantifier_expression);
DERIVE
  SELF\quantifier_expression.variables : LIST [1:?] OF UNIQUE generic_variable :=
    remove_first(SELF\multiple_arity_generic_expression.operands);
  expr : generic_expression := SELF\multiple_arity_generic_expression.operands[1];
WHERE
  WR1: SIZEOF (QUERY ( operand <*
       SELF\multiple_arity_generic_expression.operands | NOT (
       has_values_space( operand)))) = 0;
END_ENTITY;
(*
```

Attribute definitions:

SELF\quantifier_expression.variables: The list of variables, in order, which are to be treated as inputs for the constructed function.

expr: The generic expression to which function abstraction is to be applied in order to construct a function.

© ISO 2002 — All rights reserved

Formal propositions:

WR1: All the operands of instances of this type shall be mathematical expressions, as defined by function **has_values_space**.

NOTE Normally, the variables listed in **variables** are free variables in **expr** and become bound by this operation. However, it is not required that the variables appear in the expression at all.

EXAMPLE Consider the informal mathematical definition:

Let f(x) = c, where x is a real number and c is the string 'what?'.

This function can be represented as an **abstracted_expression_function** as follows:

```
LOCAL
  c : string_literal := make_string_literal('what?');
  x : maths_variable := make_maths_real_variable(the_reals, 'x');
  f : abstracted_expression_function := make_abstracted_expression_function([c, x]);
END_LOCAL;
```

4.5.71 expression_denoted_function

This type of **maths_function** and **unary_generic_expression** recognizes an expression which denotes a mathematical function as an instance of **maths_function**. Such an expression could arise, for example, as an instance of **function_application**. Suitable expressions are identified by the output of the **values_space_of** function being a **function_space**.

The domain and range of an instance of **expression_denoted_function** are derived from the **domain_argument** and **range_argument** attributes of the function space produced by **values_space_of** (**expr**).

EXPRESS specification:

```
*)
ENTITY expression_denoted_function
  SUBTYPE OF (maths_function, unary_generic_expression);
DERIVE
  expr : generic_expression := SELF\unary_generic_expression.operand;
WHERE
  WR1: (schema_prefix + 'FUNCTION_SPACE') IN TYPEOF (values_space_of(expr));
END_ENTITY;
(*
```

Attribute definitions:

expr: The expression which is to be recognized as denoting a function.

Formal propositions:

WR1: The expression shall, in fact, denote a function.

 © ISO 2002 — All rights reserved

EXAMPLE Consider the informal mathematical definition for the one-parameter family F of rotation matrices in the plane:

$$F(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

In order to recognize the expression $F(\pi/6)$ as an instance which could be used as the matrix-valued **operand** attribute of, say, a **homogeneous_linear_function**, it must first be recognized as an instance of **maths_function**. As constructed, the instance representing $F(\pi/6)$ is an instance of **function_application**. The entity type **function_application** cannot be a subtype of **maths_function** because many of its instances do not represent mathematical functions. Instead, the nature of this particular instance of **function_application** is recognized by wrapping it as an **expression_denoted_function** instance as follows:

```
LOCAL
  theta : maths_real_variable := make_maths_real_variable(the_reals, 'theta');
  cos_expr : cos_expression := make_cos_expression(theta);
  sin_expr : sin_expression := make_sin_expression(theta);
  neg_sin_expr : unary_minus_expression := make_unary_minus_expression(sin_expr);
  source : listed_data :=
    make_listed_data(1, [cos_expr, neg_sin_expr, sin_expr, cos_expr], the_reals);
  matrix : standard_table_function :=
    make_standard_table_function(1, [2,2], source, 1, by_rows);
  big_f : abstracted_expression_function :=
    make_abstracted_expression_function([matrix, theta]);
  pi_over_6 : real_literal := make_real_literal( pi/6.0);
  big_f_of_pi_over_6 : function_application :=
    make_function_application(big_f, [pi_over_6]);
  big_f_of_pi_over_6_matrix : expression_denoted_function :=
    make_expression_denoted_function(big_f_of_pi_over_6);
END_LOCAL;
```

Recall that **make**_*type_name* is a function which constructs an instance of *type_name* from the list of arguments corresponding to the attributes which would appear in an ISO-10303-21 format file for an instance of that type, with two modifications: the arguments corresponding to asterisks (derived attributes) are omitted, and the arguments corresponding to '$' signs (omitted attributes) are replaced with the indeterminate value ('?').

4.5.72 imported_point_function

This type of **maths_function** and **generic_literal** represents the constant function corresponding to a **point** instance from the **geometry_schema**. It enables instances of **point** to participate in the construction of other instances of **maths_function**.

The domain of an instance of **imported_point_function** is the space $(R^0)^1$. This implies that the function takes a single input which is an element of the space R^0. The only element of the space R^0 is the zero-tuple of real numbers, which is represented in EXPRESS by the empty list of reals.

EXPRESS specification:

```
*)
ENTITY imported_point_function
  SUBTYPE OF (maths_function, generic_literal);
  geometry : point;
END_ENTITY;
(*
```

© ISO 2002 — All rights reserved

Attribute definitions:

geometry: The geometric point whose corresponding constant function is represented.

4.5.73 imported_curve_function

This type of **maths_function** and **generic_literal** represents the parametric function defined for a **curve** instance from the **geometry_schema**. It permits non-geometric data to be associated with the curve using the curve's parametrization.

The range of an **imported_curve_function** is derived to be either $(R^2)^1$ or $(R^3)^1$ depending on the geometric dimension of the curve.

EXPRESS specification:

```
*)
ENTITY imported_curve_function
  SUBTYPE OF (maths_function, generic_literal);
  geometry          : curve;
  parametric_domain : tuple_space;
WHERE
  WR1: expression_is_constant(parametric_domain);
END_ENTITY;
(*
```

Attribute definitions:

geometry: The geometric curve whose parametric function is represented.

parametric_domain: The domain of the curve's parametric function.

Formal propositions:

WR1: The value of **parametric_domain** is a constant expression.

Informal propositions:

IP1: The value of **parametric_domain** represents the domain of the parametric function of the geometry.

4.5.74 imported_surface_function

This type of **maths_function** and **generic_literal** represents the parametric function defined for a **surface** instance from the **geometry_schema**. It permits non-geometric data to be associated with the surface using the surface's parametrization.

The range of an **imported_surface_function** is derived to be $(R^3)^1$.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
ENTITY imported_surface_function
  SUBTYPE OF (maths_function, generic_literal);
  geometry          : surface;
  parametric_domain : tuple_space;
WHERE
  WR1: expression_is_constant(parametric_domain);
END_ENTITY;
(*
```

Attribute definitions:

geometry: The geometric surface whose parametric function is represented.

parametric_domain: The domain of the surface's parametric function.

Formal propositions:

WR1: The value of **parametric_domain** is a constant expression.

Informal propositions:

IP1: The value of **parametric_domain** represents the domain of the parametric function of the geometry.

4.5.75 imported_volume_function

This type of **maths_function** and **generic_literal** represents the parametric function defined for a **volume** instance from the **geometry_schema**. It permits non-geometric data to be associated with the volume using the volume's parametrization.

The range of an **imported_surface_function** is derived to be $(R^3)^1$.

EXPRESS specification:

```
*)
ENTITY imported_volume_function
  SUBTYPE OF (maths_function, generic_literal);
  geometry          : volume;
  parametric_domain : tuple_space;
WHERE
  WR1: expression_is_constant(parametric_domain);
END_ENTITY;
(*
```

© ISO 2002 — All rights reserved

Attribute definitions:

geometry: The geometric volume whose parametric function is represented.

parametric_domain: The domain of the volume's parametric function.

Formal propositions:

WR1: The value of **parametric_domain** is a constant expression.

Informal propositions:

IP1: The value of **parametric_domain** represents the domain of the parametric function of the geometry.

4.5.76 application_defined_function

This type of **maths_function** provides for extending the schema with representations for additional types of mathematical functions.

NOTE 1 Extensions may be accomplished by defining subtypes of **application_defined_function**. In this case, the name chosen for the subtype shall identify the class of mathematical functions which are to be represented. Any parameters necessary to identify particular members of the class can be included by specializing the type of the **parameters** attribute or by adding rules to restrict the types or values of particular members of the list. Types defined in some other schema cannot be directly accommodated, but, in so far as they are constructed from aggregates of simple types, equivalent data can be stored without requiring additional attributes in a subtype declaration. In the case that no parameters are required, the empty list can be specified as the value of **parameters**.

NOTE 2 In order to integrate application-defined functions with the rest of this schema, it is required that each instance provide its domain and range spaces. This may be accomplished either by explicitly providing values for the two attributes, or by deriving appropriate values from other information as part of defining a subtype.

NOTE 3 In order to integrate application-defined functions with the expressions structure inherited from **ISO13584_generic_expressions**, it is necessary that an introduced subtype also explicitly subtype itself from one of the first or second tier subtypes of **generic_expression**, and, if subtyped from other than **simple_generic_expression**, either derive the inherited **operand(s)** attribute or explicitly provide its value.

NOTE 4 All instantiable subtypes of **maths_function** defined in this schema are associated with well-defined mathematical algorithms for obtaining outputs from inputs. The differences in results on different systems should be small. However, it is quite possible to use the application-defined function mechanism to create representations for functions which are mathematically well-defined, but for which no computationally reliable algorithm is known. For example, consider the minimum distance function on the images of two parametric surfaces. Each implementor will produce some approximation of the mathematical function, but no standard algorithm exists. Application developers must carefully consider the meaning and consequences of exchanging an instance representing such a function.

EXPRESS specification:

```
*)
ENTITY application_defined_function
```

 © ISO 2002 — All rights reserved

```
  SUBTYPE OF (maths_function);
  explicit_domain : tuple_space;
  explicit_range  : tuple_space;
  parameters      : LIST OF maths_value;
WHERE
  WR1: expression_is_constant(explicit_domain);
  WR2: expression_is_constant(explicit_range);
END_ENTITY;
(*
```

Attribute definitions:

explicit_domain: The value to be used as the domain of the function.

explicit_range: The value to be used as the range of the function.

parameters: The parameters, if any, defining the function. When no parameters are required, the empty list shall be used.

Formal propositions:

WR1: The domain shall be a constant expression.

WR2: The range shall be a constant expression.

NOTE 5 The expressions defining the domains and ranges of all functions defined in this schema are prevented from containing variables and thereby denoting indeterminate spaces. This technical restriction does not interfere with the contemplated applications and makes it practical to implement more of the semantic rules in Express. Some future edition of this part of ISO 10303 may choose to relax these requirements.

Informal propositions:

IP1: The total information associated with an instance of **application_defined_function** shall be sufficient to uniquely identify a mathematical function.

IP2: The space supplied as the domain of the function shall be compatible with the actual mathematical domain of the function.

IP3: The space supplied as the range of the function shall represent a superspace of the actual mathematical range of the function.

NOTE 6 The total information associated with an instance includes its actual subtype, the values of its attributes, and any instances of **mathematical_description** referencing it via the **described** attribute.

4.5.77 mathematical_description

This type relates a **maths_expression** and a description or definition of the same mathematical object or expression by a string in an identified language or identified encoding of an identified language.

NOTE 1 A value of type **maths_expression** may be an instance of **maths_space** or an instance of **maths_function** or an instance of **generic_expression** or a simple data value. The most likely use of the **mathematical_description** entity, however, is to define an instance of **application_defined_function**

© ISO 2002 — All rights reserved

using a language other than EXPRESS.

NOTE 2 Examples of appropriate languages are computer programming languages and the Mathematical Markup Language (MathML).

MathML is being developed by the HTML–Math Working Group of the World-Wide Web Consortium (W3C). A list of current W3C Technical Reports can be found at `<http://www.w3.org/TR>`. The edition examined during the development of this part of ISO 10303 is listed in the Bibliography as [2]. The latest edition at the time of final editing of this part of ISO 10303 was [3]. The work of the OpenMath Society at `<http://www.openmath.org>` may also be of interest in this context.

NOTE 3 An application protocol or application module may constrain the allowable languages or language encodings and specify the labels to be used for each.

NOTE 4 Languages using a different character set than that used by Express, such as MathML's use of UNICODE, require special care. In some cases, it may be necessary or desirable to specify a special encoding of conflicting or missing elements into an Express string value.

EXPRESS specification:

```
*)
ENTITY mathematical_description;
  described  : maths_expression;
  describing : STRING;
  encoding   : label;
END_ENTITY;
(*
```

Attribute definitions:

described: The mathematical object or expression being described.

describing: The description, as an Express string value.

encoding: The identification of the language or encoding used in making the description.

Informal propositions:

IP1: The value of attribute **describing** shall be a valid instance of the language or encoding identified by the value of attribute **encoding**.

IP2: In so far as the value of the attribute **described** is already defined in this part of ISO 10303, the meaning of the value of **describing** shall be consistent with it.

4.6 Mathematical functions schema function definitions

4.6.1 all_members_of_es

This function determines whether all members of a list of **maths_value**s are members of an elementary space identified by a value of **elementary_space_enumerators**.

NOTE This function is used by the **subspace_of_es** function.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION all_members_of_es(sv : SET OF maths_value;
                           es : elementary_space_enumerators) : LOGICAL;
  CONSTANT
    base_types : SET OF STRING := ['NUMBER','COMPLEX_NUMBER_LITERAL','REAL',
      'INTEGER','LOGICAL','BOOLEAN','STRING','BINARY','MATHS_SPACE',
      'MATHS_FUNCTION','LIST','ELEMENTARY_SPACE_ENUMERATORS','ORDERING_TYPE',
      'LOWER_UPPER','SYMMETRY_TYPE','ELEMENTARY_FUNCTION_ENUMERATORS',
      'OPEN_CLOSED','SPACE_CONSTRAINT_TYPE','REPACKAGE_OPTIONS',
      'EXTENSION_OPTIONS'];
  END_CONSTANT;
  LOCAL
    v : maths_value;
    key_type : STRING := '';
    types : SET OF STRING;
    ge : generic_expression;
    cum : LOGICAL := TRUE;
    vspc : maths_space;
  END_LOCAL;
  IF NOT EXISTS (sv) OR NOT EXISTS (es) THEN  RETURN (FALSE);  END_IF;
  CASE es OF
  es_numbers :          key_type := 'NUMBER';
  es_complex_numbers : key_type := 'COMPLEX_NUMBER_LITERAL';
  es_reals :            key_type := 'REAL';
  es_integers :         key_type := 'INTEGER';
  es_logicals :         key_type := 'LOGICAL';
  es_booleans :         key_type := 'BOOLEAN';
  es_strings :          key_type := 'STRING';
  es_binarys :          key_type := 'BINARY';
  es_maths_spaces :     key_type := 'MATHS_SPACE';
  es_maths_functions : key_type := 'MATHS_FUNCTION';
  es_generics :         RETURN (TRUE);
  END_CASE;
  REPEAT i := 1 TO SIZEOF (sv);
    IF NOT EXISTS (sv[i]) THEN  RETURN (FALSE);  END_IF;
    v := simplify_maths_value(sv[i]);
    types := stripped_typeof(v);
    IF key_type IN types THEN  SKIP;  END_IF;
    IF (es = es_numbers) AND ('COMPLEX_NUMBER_LITERAL' IN types) THEN  SKIP;  END_IF;
    IF SIZEOF (base_types * types) > 0 THEN  RETURN (FALSE);  END_IF;
    -- Must be a generic_expression which doesn't simplify and which is not a
    -- complex_number_literal, maths_space, or maths_function.
    ge := v;
    IF has_values_space(ge) THEN
      vspc := values_space_of(ge);
      IF NOT subspace_of_es(vspc,es) THEN
        IF NOT compatible_spaces(vspc,make_elementary_space(es)) THEN
          RETURN (FALSE);
        END_IF;
        cum := UNKNOWN;
      END_IF;
    ELSE
      cum := UNKNOWN;
    END_IF;
    IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
  END_REPEAT;
  RETURN (cum);
```

© ISO 2002 — All rights reserved

```
END_FUNCTION;  -- all_members_of_es
(*
```

Argument definitions:

lv: (input) The list of **maths_value**s.

es: (input) The enumberation value identifying the elementary space.

return: (output) A LOGICAL value which is TRUE if all members of the list can be determined to belong to the indicated elementary space, FALSE if at least one member can be determined not to belong, and UNKNOWN, otherwise.

4.6.2 any_space_satisfies

This function determines whether a pair consisting of a value of **space_constraint_type** and a value of **maths_space** describe a constraint that is satisfied by all mathematical spaces.

NOTE This function is used in connection with **function_space** instances.

EXPRESS specification:

```
*)
FUNCTION any_space_satisfies(sc  : space_constraint_type;
                             spc : maths_space) : BOOLEAN;
  LOCAL
    spc_id : elementary_space_enumerators;
  END_LOCAL;
  IF (sc = sc_equal) OR NOT ('ELEMENTARY_SPACE' IN stripped_typeof(spc)) THEN
    RETURN (FALSE);
  END_IF;
  spc_id := spc\elementary_space.space_id;
  IF sc = sc_subspace THEN
    RETURN (bool(spc_id = es_generics));
  END_IF;
  IF sc = sc_member THEN
    RETURN (bool((spc_id = es_generics) OR (spc_id = es_maths_spaces)));
  END_IF;
  -- Should be unreachable.
  RETURN (?);
END_FUNCTION;  -- any_space_satisfies
(*
```

Argument definitions:

sc: (input) The value of **space_constraint_type**.

spc: (input) The **maths_space**.

return: (output) The BOOLEAN value indicating whether any mathematical space satisfies the implied constraint or not.

 © ISO 2002 — All rights reserved

4.6.3 assoc_product_space

This function returns an instance of **tuple_space** representing the associative cartesian product of its two **tuple_space** arguments.

EXPRESS specification:

```
*)
FUNCTION assoc_product_space(ts1, ts2 : tuple_space) : tuple_space;
  LOCAL
    types1 : SET OF STRING := stripped_typeof (ts1);
    types2 : SET OF STRING := stripped_typeof (ts2);
    up1, up2 : uniform_product_space := make_uniform_product_space(the_reals,1);
    lp1, lp2, lps : listed_product_space := the_zero_tuple_space;
    et1, et2, ets : extended_tuple_space := the_tuples;
    use_up1, use_up2, use_lp1, use_lp2 : BOOLEAN;
    factors : LIST OF maths_space := [];
    tspace : tuple_space;
  END_LOCAL;
  -- Identify type of first operand
  IF 'UNIFORM_PRODUCT_SPACE' IN types1 THEN
    up1 := ts1;  use_up1 := true;  use_lp1 := false;
  ELSE
    IF 'LISTED_PRODUCT_SPACE' IN types1 THEN
      lp1 := ts1;  use_up1 := false;  use_lp1 := true;
    ELSE
      IF NOT ('EXTENDED_TUPLE_SPACE' IN types1) THEN
        -- Unreachable when this function was written.
        RETURN (?);
      END_IF;
      et1 := ts1;  use_up1 := false;  use_lp1 := false;
    END_IF;
  END_IF;
  -- Identify type of second operand
  IF 'UNIFORM_PRODUCT_SPACE' IN types2 THEN
    up2 := ts2;  use_up2 := true;  use_lp2 := false;
  ELSE
    IF 'LISTED_PRODUCT_SPACE' IN types2 THEN
      lp2 := ts2;  use_up2 := false;  use_lp2 := true;
    ELSE
      IF NOT ('EXTENDED_TUPLE_SPACE' IN types2) THEN
        -- Unreachable when this function was written.
        RETURN (?);
      END_IF;
      et2 := ts2;  use_up2 := false;  use_lp2 := false;
    END_IF;
  END_IF;
  -- Construction for each combination of cases
  IF use_up1 THEN
    IF use_up2 THEN
      IF up1.base = up2.base THEN
        tspace := make_uniform_product_space(up1.base, up1.exponent + up2.exponent);
      ELSE
        factors := [up1.base : up1.exponent, up2.base : up2.exponent];
        tspace := make_listed_product_space(factors);
      END_IF;
    ELSE
```

© ISO 2002 — All rights reserved

```
      IF use_lp2 THEN
        -- Avoid compiler confusion by breaking into two lines.
        factors := [up1.base : up1.exponent];
        factors := factors + lp2.factors;
        tspace := make_listed_product_space(factors);
      ELSE
        tspace := assoc_product_space(up1, et2.base);
        tspace := make_extended_tuple_space(tspace, et2.extender);
      END_IF;
    END_IF;
  ELSE
    IF use_lp1 THEN
      IF use_up2 THEN
        -- Avoid compiler confusion by breaking into two lines.
        factors := [up2.base : up2.exponent];
        factors := lp1.factors + factors;
        tspace := make_listed_product_space(factors);
      ELSE
        IF use_lp2 THEN
          tspace := make_listed_product_space(lp1.factors + lp2.factors);
        ELSE
          tspace := assoc_product_space(lp1, et2.base);
          tspace := make_extended_tuple_space(tspace, et2.extender);
        END_IF;
      END_IF;
    ELSE
      IF use_up2 THEN
        IF et1.extender = up2.base THEN
          tspace := assoc_product_space(et1.base, up2);
          tspace := make_extended_tuple_space(tspace, et1.extender);
        ELSE
          -- No subtype is available to represent this cartesian product.
          RETURN (?);
        END_IF;
      ELSE
        IF use_lp2 THEN
          factors := lp2.factors;
          REPEAT i := 1 TO SIZEOF (factors);
            IF et1.extender <> factors[i] THEN
              -- No subtype available to represent this cartesian product.
              RETURN (?);
            END_IF;
          END_REPEAT;
          tspace := assoc_product_space(et1.base, lp2);
          tspace := make_extended_tuple_space(tspace, et1.extender);
        ELSE
          IF et1.extender = et2.extender THEN
            -- Next line may assign indeterminate (?) to tspace.
            tspace := assoc_product_space(et1, et2.base);
          ELSE
            -- No subtype available to represent this cartesian product.
            RETURN (?);
          END_IF;
        END_IF;
      END_IF;
    END_IF;
  END_IF;
  RETURN (tspace);
END_FUNCTION;  -- assoc_product_space
```

 © ISO 2002 — All rights reserved

```
(*
```

Argument definitions:

sp1: (input) The first tuple space operand.

sp2: (input) The second tuple space operand.

return: (output) An instance of **tuple_space** representing the associative cartesian product of the two operands.

4.6.4 atan2

This function computes the direction angle to the point in the plane defined by its two real arguments. Values returned lie in the range $-\pi < r \leq \pi$. The direction angle is the signed radian measure of the angle between the positive x-axis and the ray emanating from the origin and passing through the point. Points on the positive y-axis have direction angle $\pi/2$. In accordance with tradition, the y-coordinate of the point is the first argument, and the x-coordinate is the second. This function returns the indeterminate value when the point is the origin.

NOTE This is the traditional "atan2" function. Unlike the traditional "atan" ("arc tangent" or "inverse tangent") function, it takes two arguments and distinguishes among all four quadrants. The argument order probably derived from thinking of it as an extension of "atan(y/x)". The EXPRESS **atan** function is a peculiar hybrid of the traditional "atan" and "atan2" functions.

EXPRESS specification:

```
*)
FUNCTION atan2(y, x : REAL) : REAL;
  LOCAL
    r : REAL;
  END_LOCAL;
  IF (y = 0.0) AND (x = 0.0) THEN  RETURN (?);  END_IF;
  r := atan(y,x);
  IF x < 0.0 THEN
    IF y < 0.0 THEN  r := r - PI;
    ELSE             r := r + PI;  END_IF;
  END_IF;
  RETURN (r);
END_FUNCTION;  -- atan2
(*
```

Argument definitions:

y: (input) The y coordinate of the point.

x: (input) The x coordinate of the point

return: (output) The direction angle to the point.

© ISO 2002 — All rights reserved

4.6.5 bool

This function converts Express LOGICAL values into BOOLEAN values by turning any value which is not TRUE into FALSE.

EXPRESS specification:

```
*)
FUNCTION bool(lgcl: LOGICAL) : BOOLEAN;
  IF NOT EXISTS (lgcl) THEN  RETURN (FALSE);  END_IF;
  IF lgcl <> TRUE      THEN  RETURN (FALSE);  END_IF;
  RETURN (TRUE);
END_FUNCTION;  -- bool
(*
```

Argument definitions:

lgcl: (input) The logical value.

return: (output) The boolean value TRUE when the input is TRUE and FALSE, otherwise.

4.6.6 check_sparse_index_domain

This function verifies that the domain of the index function in an instance of **basic_sparse_matrix** is consistent with the other attributes.

EXPRESS specification:

```
*)
FUNCTION check_sparse_index_domain(idxdom : tuple_space;
                                   base   : zero_or_one;
                                   shape  : LIST [1:?] OF positive_integer;
                                   order  : ordering_type) : BOOLEAN;
  LOCAL
    mthspc : maths_space;
    interval : finite_integer_interval;
    i : INTEGER;
  END_LOCAL;
  mthspc := factor1(idxdom);
  -- A consequence of WR1 of basic_sparse_matrix is that here we need only
  -- consider the case that mthspc is a finite integer interval and is the only
  -- factor space of idxdom.
  interval := mthspc;
  IF order = by_rows THEN  i := 1;  ELSE  i := 2;  END_IF;
  RETURN (bool((interval.min <= base) AND (interval.max >= base + shape[i])));
  -- The index function is evaluated at (base+shape[i]) when determining the
  -- upper search bound for entries of the last row or column, respectively.
END_FUNCTION;  -- check_sparse_index_domain;
(*
```

 © ISO 2002 — All rights reserved

Argument definitions:

idxdom: (input) The domain of the basic sparse matrix index function.

base: (input) The index base of the basic sparse matrix.

shape: (input) The shape of the basic sparse matrix.

order: (input) The ordering of the basic sparse matrix.

return: (output) The boolean value indicating whether the domain of the index function is consistent with the other attributes.

4.6.7 check_sparse_loc_range

This function verifies that the range of the location function in an instance of **basic_sparse_-matrix** is consistent with the other attributes.

EXPRESS specification:

```
*)
FUNCTION check_sparse_loc_range(locrng : tuple_space;
                                base   : zero_or_one;
                                shape  : LIST [1:?] OF positive_integer;
                                order  : ordering_type) : BOOLEAN;
  LOCAL
    mthspc : maths_space;
    interval : finite_integer_interval;
    i : INTEGER;
  END_LOCAL;
  IF space_dimension(locrng) <> 1 THEN  RETURN (FALSE);  END_IF;
  mthspc := factor1(locrng);
  IF NOT ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (mthspc)) THEN
    RETURN (FALSE);
  END_IF;
  interval := mthspc;
  IF order = by_rows THEN  i := 2;  ELSE  i := 1;  END_IF;
  RETURN (bool((interval.min >= base) AND (interval.max <= base + shape[i] - 1)));
END_FUNCTION;  -- check_sparse_loc_range;
(*
```

Argument definitions:

locrng: (input) The range of the basic sparse matrix location function.

base: (input) The index base of the basic sparse matrix.

shape: (input) The shape of the basic sparse matrix.

order: (input) The ordering of the basic sparse matrix.

return: (output) The boolean value indicating whether the range of the location function is consistent with the other attributes.

© ISO 2002 — All rights reserved

4.6.8 check_sparse_index_to_loc

This function is used in WR5 in **basic_sparse_matrix** to verify the compatibility of two of its operands. The test is whether the integer range interval of the function used as the value of attribute **index** lies within the integer domain interval of the function used as the value of attribute **loc**, except perhaps at its maximum value. This function assumes automatic repackaging between integers and one-tuples of integers will be performed as needed in any evaluator for **basic_sparse_matrix**.

NOTE This rather special requirement follows from the evaluation algorithm for **basic_sparse_matrix**.

EXPRESS specification:

```
*)
FUNCTION check_sparse_index_to_loc(index_range, loc_domain : tuple_space) : BOOLEAN;
  LOCAL
    temp : maths_space;
    idx_rng_itvl, loc_dmn_itvl : finite_integer_interval;
  END_LOCAL;
  temp := factor1 (index_range);
  IF (schema_prefix + 'TUPLE_SPACE') IN TYPEOF (temp) THEN
    temp := factor1 (temp);
  END_IF;
  IF NOT ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (temp)) THEN
    RETURN (FALSE);
  END_IF;
  idx_rng_itvl := temp;
  temp := factor1 (loc_domain);
  IF (schema_prefix + 'TUPLE_SPACE') IN TYPEOF (temp) THEN
    temp := factor1 (temp);
  END_IF;
  IF NOT ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (temp)) THEN
    RETURN (FALSE);
  END_IF;
  loc_dmn_itvl := temp;
  RETURN (bool((loc_dmn_itvl.min <= idx_rng_itvl.min) AND
    (idx_rng_itvl.max <= loc_dmn_itvl.max+1)));
END_FUNCTION;  -- check_sparse_index_to_loc
(*
```

Argument definitions:

index_range: (input) The range space of the function serving as the value of the **index** attribute of an instance of **basic_sparse_matrix**.

loc_domain: (input) The domain space of the function serving as the value of the **loc** attribute of an instance of **basic_sparse_matrix**.

return: (output) A BOOLEAN value indicating whether the two spaces are suitable for these uses in an instance of **basic_sparse_matrix**.

 © ISO 2002 — All rights reserved

4.6.9 compare_basis_and_coef

This function verifies the consistency of the **basis** and **coef** attributes in a **b_spline_function** instance. In particular, it verifies theat the dimension of the coefficient table is at least as great as the number of B-spline bases and that the numbers of basis functions in each B-spline basis entity matches the numbers of subscripts in the corresponding dimension of the coefficient table.

EXPRESS specification:

```
*)
FUNCTION compare_basis_and_coef(basis : LIST [1:?] OF b_spline_basis;
                                coef  : maths_function) : BOOLEAN;
  LOCAL
    shape : LIST OF positive_integer;
  END_LOCAL;
  IF NOT EXISTS (basis) OR NOT EXISTS (coef) THEN  RETURN (FALSE);  END_IF;
  shape := shape_of_array(coef);
  IF NOT EXISTS (shape) THEN  RETURN (FALSE);  END_IF;
  IF SIZEOF (shape) < SIZEOF (basis) THEN  RETURN (FALSE);  END_IF;
  REPEAT i := 1 TO SIZEOF (basis);
    IF (basis[i].num_basis = shape[i]) <> TRUE THEN  RETURN (FALSE);  END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- compare_basis_and_coef
(*
```

Argument definitions:

basis: (input) The list of B-spline bases.

coef: (input) The table function containing the B-spline coefficients.

return: (output) A BOOLEAN value which is TRUE if the numbers of basis functions in the input B-spline bases and the numbers of subscripts in the corresponding dimensions of the coefficient table all match.

4.6.10 compare_list_and_value

This function compares a list and a value, and returns TRUE only if the comparison is TRUE between every list member and the given value. In every other case, the function returns FALSE.

NOTE This function is used by **selection_function** and **selection_insertion_function**.

EXPRESS specification:

```
*)
FUNCTION compare_list_and_value(lv : LIST OF GENERIC:G;
                                op : elementary_function_enumerators;
                                v  : GENERIC:G) : BOOLEAN;
  IF NOT EXISTS (lv) OR NOT EXISTS (op) OR NOT EXISTS (v) THEN
```

© ISO 2002 — All rights reserved

```
    RETURN (FALSE);
  END_IF;
  REPEAT i := 1 TO SIZEOF (lv);
    IF NOT compare_values(lv[i], op, v) THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- compare_list_and_value
(*
```

Argument definitions:

lv: (input) The list of values to use in the comparison.

op: (input) The comparison operation to use using only the range from **ef_eq_i** to **ef_le_i** from **elementary_function_enumerators**.

v: The value to use in the comparison.

return: (output) A BOOLEAN value which is TRUE if every member of the input list stands in the indicated relation to the given value **v**.

4.6.11 compare_values

This function compares two values in accordance with the indicated comparison operator, and returns TRUE only if the comparison is true. In every other case, the function returns FALSE.

EXPRESS specification:

```
*)
FUNCTION compare_values(v1 : GENERIC:G;
                        op : elementary_function_enumerators;
                        v2 : GENERIC:G) : BOOLEAN;
  -- This algorithm assumes a comparison between "incompatible" types will
  -- produce the indeterminate value (or UNKNOWN?).
  LOCAL
    logl : LOGICAL := UNKNOWN;
  END_LOCAL;
  IF NOT EXISTS (v1) OR NOT EXISTS (op) OR NOT EXISTS (v2) THEN
    RETURN (FALSE);
  END_IF;
  CASE op OF
  ef_eq_i : logl := (v1 = v2);
  ef_ne_i : logl := (v1 <> v2);
  ef_gt_i : logl := (v1 > v2);
  ef_lt_i : logl := (v1 < v2);
  ef_ge_i : logl := (v1 >= v2);
  ef_le_i : logl := (v1 <= v2);
  END_CASE;
  IF EXISTS (logl) THEN
    IF logl = TRUE THEN  RETURN (TRUE);  END_IF;
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- compare_values
(*
```


© ISO 2002 — All rights reserved

Argument definitions:

v1: (input) The first value to use in the comparison.

op: (input) The comparison operation to use.

v2: (input) The second value to use in the comparison.

return: (output) A BOOLEAN value which is TRUE if the comparison between the values **v1** and **v2** indicated by **op** is true.

4.6.12 compatible_complex_number_regions

This function defines whether two instances of **maths_space** which happen to be complex number subsets are compatible or not. See **compatible_spaces**.

EXPRESS specification:

```
*)
FUNCTION compatible_complex_number_regions(sp1, sp2 : maths_space) : BOOLEAN;
  LOCAL
    typenames : SET OF string := stripped_typeof (sp1);
    crgn1, crgn2 : cartesian_complex_number_region;
    prgn1, prgn2, prgn1c2, prgn2c1 : polar_complex_number_region;
    sp1_is_crgn, sp2_is_crgn : BOOLEAN;
  END_LOCAL;
  IF 'CARTESIAN_COMPLEX_NUMBER_REGION' IN typenames THEN
    sp1_is_crgn := TRUE;
    crgn1 := sp1;
  ELSE
    IF 'POLAR_COMPLEX_NUMBER_REGION' IN typenames THEN
      sp1_is_crgn := FALSE;
      prgn1 := sp1;
    ELSE
      -- Improper usage: Default response is to assume compatibility.
      RETURN (TRUE);
    END_IF;
  END_IF;
  typenames := stripped_typeof (sp2);
  IF 'CARTESIAN_COMPLEX_NUMBER_REGION' IN typenames THEN
    sp2_is_crgn := TRUE;
    crgn2 := sp2;
  ELSE
    IF 'POLAR_COMPLEX_NUMBER_REGION' IN typenames THEN
      sp2_is_crgn := FALSE;
      prgn2 := sp2;
    ELSE
      -- Improper usage: Default response is to assume compatibility.
      RETURN (TRUE);
    END_IF;
  END_IF;
  IF sp1_is_crgn AND sp2_is_crgn THEN
    -- two cartesian regions
    RETURN (compatible_intervals(crgn1.real_constraint, crgn2.real_constraint)
      AND compatible_intervals(crgn1.imag_constraint, crgn2.imag_constraint));
  END_IF;
  IF NOT sp1_is_crgn AND NOT sp2_is_crgn AND
```

© ISO 2002 — All rights reserved

```
    (prgn1.centre.real_part = prgn2.centre.real_part) AND
    (prgn1.centre.imag_part = prgn2.centre.imag_part) THEN
    -- two polar regions with common centre
    IF NOT compatible_intervals(prgn1.distance_constraint,
      prgn2.distance_constraint) THEN
      RETURN (FALSE);
    END_IF;
    IF compatible_intervals(prgn1.direction_constraint,
      prgn2.direction_constraint) THEN
      RETURN (TRUE);
    END_IF;
    -- Deal with direction ambiguity by 2 pi.
    IF (prgn1.direction_constraint.max > PI) AND (prgn2.direction_constraint.max < PI)
      THEN
      RETURN (compatible_intervals(prgn2.direction_constraint,
        make_finite_real_interval(-PI,open,prgn1.direction_constraint.max-2.0*PI,
        prgn1.direction_constraint.max_closure)));
    END_IF;
    IF (prgn2.direction_constraint.max > PI) AND (prgn1.direction_constraint.max < PI)
      THEN
      RETURN (compatible_intervals(prgn1.direction_constraint,
        make_finite_real_interval(-PI,open,prgn2.direction_constraint.max-2.0*PI,
        prgn2.direction_constraint.max_closure)));
    END_IF;
    RETURN (FALSE);
  END_IF;
  -- Make do with imperfect tests for remaining cases.
  IF sp1_is_crgn AND NOT sp2_is_crgn THEN
    crgn2 := enclose_pregion_in_cregion(prgn2);
    prgn1 := enclose_cregion_in_pregion(crgn1,prgn2.centre);
    RETURN (compatible_complex_number_regions(crgn1,crgn2)
      AND compatible_complex_number_regions(prgn1,prgn2));
  END_IF;
  IF NOT sp1_is_crgn AND sp2_is_crgn THEN
    crgn1 := enclose_pregion_in_cregion(prgn1);
    prgn2 := enclose_cregion_in_pregion(crgn2,prgn1.centre);
    RETURN (compatible_complex_number_regions(crgn1,crgn2)
      AND compatible_complex_number_regions(prgn1,prgn2));
  END_IF;
  -- Two polar regions with different centres
  prgn1c2 := enclose_pregion_in_pregion(prgn1,prgn2.centre);
  prgn2c1 := enclose_pregion_in_pregion(prgn2,prgn1.centre);
  RETURN (compatible_complex_number_regions(prgn1,prgn2c1)
    AND compatible_complex_number_regions(prgn1c2,prgn2));
END_FUNCTION;  -- compatible_complex_number_regions
(*
```

Argument definitions:

sp1: (input) The first space.

sp2: (input) The second space.

return: (output) A BOOLEAN value of FALSE is returned if it is determined that the intersection of the two spaces is empty. Otherwise, TRUE is returned.

NOTE One can construct regions which the preceding function reports as compatible, but which, in fact, have empty intersection. This is acceptable. The consequence is merely that some valid instances

 © ISO 2002 — All rights reserved

have no practical use.

4.6.13 compatible_es_values

This function ascertains whether two **elementary_space_enumerators** values identify two compatible spaces. For the purposes of this part of ISO 10303, the integers, the real numbers, and the complex numbers shall be treated as disjoint spaces whose union is the space of all numbers.

NOTE The Express language considers the `INTEGER` type to be a specialization of the `REAL` type. This is inconsistent with all programming languages and with most of mathematics. The normal mathematical relationship is that there is a natural isomorphism between the integers and a unique subset of the real numbers, but they are not identical. The usual conversions between integers and reals are applications of this isomorphism and its inverse.

EXPRESS specification:

```
*)
FUNCTION compatible_es_values(esval1, esval2 : elementary_space_enumerators) : BOOLEAN;
  LOCAL
    esval1_is_numeric, esval2_is_numeric : LOGICAL;
  END_LOCAL;
  IF (esval1 = esval2) OR (esval1 = es_generics) OR (esval2 = es_generics) THEN
    RETURN (TRUE);
  END_IF;
  esval1_is_numeric := (esval1 >= es_numbers) AND (esval1 <= es_integers);
  esval2_is_numeric := (esval2 >= es_numbers) AND (esval2 <= es_integers);
  IF (esval1_is_numeric AND (esval2 = es_numbers)) OR
    (esval2_is_numeric AND (esval1 = es_numbers)) THEN
    RETURN (TRUE);
  END_IF;
  IF esval1_is_numeric XOR esval2_is_numeric THEN
    RETURN (FALSE);
  END_IF;
  IF ((esval1 = es_logicals) AND (esval2 = es_booleans)) OR
    ((esval1 = es_booleans) AND (esval2 = es_logicals)) THEN
    RETURN (TRUE);
  END_IF;
  -- All other cases are incompatible
  RETURN (FALSE);
END_FUNCTION;  -- compatible_es_values
(*
```

Argument definitions:

esval1: (input) The first **elementary_space_enumerators** value.

esval2: (input) The second **elementary_space_enumerators** value.

return: (output) A BOOLEAN value of FALSE is returned if it is determined that the intersection of the spaces identified by the values is empty. Otherwise, TRUE is returned.

© ISO 2002 — All rights reserved

4.6.14 compatible_intervals

This function determines whether two instances of **maths_space** which happen to be both integer intervals or both real intervals are compatible or not. See **compatible_spaces**.

EXPRESS specification:

```
*)
FUNCTION compatible_intervals(sp1, sp2 : maths_space) : BOOLEAN;
  LOCAL
    amin, amax : REAL;
  END_LOCAL;
  IF min_exists(sp1) AND max_exists(sp2) THEN
    amin := real_min(sp1);  amax := real_max(sp2);
    IF amin > amax THEN  RETURN (FALSE);  END_IF;
    IF amin = amax THEN
      RETURN (min_included(sp1) AND max_included(sp2));
    END_IF;
  END_IF;
  IF min_exists(sp2) AND max_exists(sp1) THEN
    amin := real_min(sp2);  amax := real_max(sp1);
    IF amin > amax THEN  RETURN (FALSE);  END_IF;
    IF amin = amax THEN
      RETURN (min_included(sp2) AND max_included(sp1));
    END_IF;
  END_IF;
  RETURN (TRUE);
END_FUNCTION;  -- compatible_intervals
(*
```

Argument definitions:

sp1: (input) The first space.

sp2: (input) The second space.

return: (output) A BOOLEAN value of FALSE is returned if it is determined that the intersection of the two spaces is empty. Otherwise, TRUE is returned.

4.6.15 compatible_spaces

This function defines whether two instances of **maths_space** are compatible. Two spaces are regarded as compatible if it is not "obvious" that their intersection is empty.

NOTE As more and more subtypes of **maths_space** are added to this schema it will become more and more difficult to ascertain whether the intersection of two of them is empty. Since this function is used only to eliminate nonsensical instances, it is sufficient to rule out the obvious cases and permit the non-obvious cases to be valid as instances, although, perhaps, still useless as objects.

EXPRESS specification:

```
*)
FUNCTION compatible_spaces(sp1, sp2 : maths_space) : BOOLEAN;
```

 © ISO 2002 — All rights reserved

```
LOCAL
  types1 : SET OF STRING := stripped_typeof (sp1);
  types2 : SET OF STRING := stripped_typeof (sp2);
  lgcl : LOGICAL := UNKNOWN;
  m, n : INTEGER;
  s1, s2 : maths_space;
END_LOCAL;
IF 'FINITE_SPACE' IN types1 THEN
  REPEAT i := 1 TO SIZEOF (sp1\finite_space.members);
    lgcl := member_of(sp1\finite_space.members[i], sp2);
    IF lgcl <> FALSE THEN
      RETURN (TRUE);
    END_IF;
  END_REPEAT;
  RETURN (FALSE);
END_IF;
IF 'FINITE_SPACE' IN types2 THEN
  REPEAT i := 1 TO SIZEOF (sp2\finite_space.members);
    lgcl := member_of(sp2\finite_space.members[i], sp1);
    IF lgcl <> FALSE THEN
      RETURN (TRUE);
    END_IF;
  END_REPEAT;
  RETURN (FALSE);
END_IF;
IF 'ELEMENTARY_SPACE' IN types1 THEN
  IF sp1\elementary_space.space_id = es_generics THEN
    RETURN (TRUE);
  END_IF;
  IF 'ELEMENTARY_SPACE' IN types2 THEN
    RETURN (compatible_es_values(sp1\elementary_space.space_id,
      sp2\elementary_space.space_id));
  END_IF;
  IF ('FINITE_INTEGER_INTERVAL' IN types2) OR
    ('INTEGER_INTERVAL_FROM_MIN' IN types2) OR
    ('INTEGER_INTERVAL_TO_MAX' IN types2) THEN
    RETURN (compatible_es_values(sp1\elementary_space.space_id, es_integers));
  END_IF;
  IF ('FINITE_REAL_INTERVAL' IN types2) OR
    ('REAL_INTERVAL_FROM_MIN' IN types2) OR
    ('REAL_INTERVAL_TO_MAX' IN types2) THEN
    RETURN (compatible_es_values(sp1\elementary_space.space_id, es_reals));
  END_IF;
  IF ('CARTESIAN_COMPLEX_NUMBER_REGION' IN types2) OR
    ('POLAR_COMPLEX_NUMBER_REGION' IN types2) THEN
    RETURN (compatible_es_values(sp1\elementary_space.space_id, es_complex_numbers));
  END_IF;
  IF 'TUPLE_SPACE' IN types2 THEN
    RETURN (FALSE);
  END_IF;
  IF 'FUNCTION_SPACE' IN types2 THEN
    RETURN (bool(sp1\elementary_space.space_id = es_maths_functions));
  END_IF;
  -- Should be unreachable.
  RETURN (TRUE);
END_IF;
IF 'ELEMENTARY_SPACE' IN types2 THEN
  IF sp2\elementary_space.space_id = es_generics THEN
    RETURN (TRUE);
```

© ISO 2002 — All rights reserved

```
    END_IF;
    IF ('FINITE_INTEGER_INTERVAL' IN types1) OR
      ('INTEGER_INTERVAL_FROM_MIN' IN types1) OR
      ('INTEGER_INTERVAL_TO_MAX' IN types1) THEN
      RETURN (compatible_es_values(sp2\elementary_space.space_id, es_integers));
    END_IF;
    IF ('FINITE_REAL_INTERVAL' IN types1) OR
      ('REAL_INTERVAL_FROM_MIN' IN types1) OR
      ('REAL_INTERVAL_TO_MAX' IN types1) THEN
      RETURN (compatible_es_values(sp2\elementary_space.space_id, es_reals));
    END_IF;
    IF ('CARTESIAN_COMPLEX_NUMBER_REGION' IN types1) OR
      ('POLAR_COMPLEX_NUMBER_REGION' IN types1) THEN
      RETURN (compatible_es_values(sp2\elementary_space.space_id, es_complex_numbers));
    END_IF;
    IF 'TUPLE_SPACE' IN types1 THEN
      RETURN (FALSE);
    END_IF;
    IF 'FUNCTION_SPACE' IN types1 THEN
      RETURN (bool(sp2\elementary_space.space_id = es_maths_functions));
    END_IF;
    -- Should be unreachable.
    RETURN (TRUE);
  END_IF;
  IF subspace_of_es(sp1,es_integers) THEN  -- Note that sp1 finite already handled.
    IF subspace_of_es(sp2,es_integers) THEN  -- Note that sp2 finite already handled.
      RETURN (compatible_intervals(sp1,sp2));
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF subspace_of_es(sp2,es_integers) THEN
    RETURN (FALSE);
  END_IF;
  IF subspace_of_es(sp1,es_reals) THEN  -- Note that sp1 finite already handled.
    IF subspace_of_es(sp2,es_reals) THEN  -- Note that sp2 finite already handled.
      RETURN (compatible_intervals(sp1,sp2));
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF subspace_of_es(sp2,es_reals) THEN
    RETURN (FALSE);
  END_IF;
  IF subspace_of_es(sp1,es_complex_numbers) THEN  -- Note sp1 finite already handled.
    IF subspace_of_es(sp2,es_complex_numbers) THEN  -- Note sp2 finite already handled.
      RETURN (compatible_complex_number_regions(sp1,sp2));
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF subspace_of_es(sp2,es_complex_numbers) THEN
    RETURN (FALSE);
  END_IF;
  IF 'UNIFORM_PRODUCT_SPACE' IN types1 THEN
    IF 'UNIFORM_PRODUCT_SPACE' IN types2 THEN
      IF sp1\uniform_product_space.exponent <> sp2\uniform_product_space.exponent THEN
        RETURN (FALSE);
      END_IF;
      RETURN (compatible_spaces(sp1\uniform_product_space.base,
        sp2\uniform_product_space.base));
    END_IF;
```

 © ISO 2002 — All rights reserved

```
    IF 'LISTED_PRODUCT_SPACE' IN types2 THEN
      n := SIZEOF (sp2\listed_product_space.factors);
      IF sp1\uniform_product_space.exponent <> n THEN
        RETURN (FALSE);
      END_IF;
      REPEAT i := 1 TO n;
        IF NOT compatible_spaces(sp1\uniform_product_space.base,
          sp2\listed_product_space.factors[i]) THEN
          RETURN (FALSE);
        END_IF;
      END_REPEAT;
      RETURN (TRUE);
    END_IF;
    IF 'EXTENDED_TUPLE_SPACE' IN types2 THEN
      m := sp1\uniform_product_space.exponent;
      n := space_dimension(sp2\extended_tuple_space.base);
      IF m < n THEN
        RETURN (FALSE);
      END_IF;
      IF m = n THEN
        RETURN (compatible_spaces(sp1, sp2\extended_tuple_space.base));
      END_IF;
      RETURN (compatible_spaces(sp1, assoc_product_space(
        sp2\extended_tuple_space.base, make_uniform_product_space(
        sp2\extended_tuple_space.extender, m - n))));
    END_IF;
    IF 'FUNCTION_SPACE' IN types2 THEN
      RETURN (FALSE);
    END_IF;
    -- Should be unreachable.
    RETURN (TRUE);
  END_IF;
  IF 'LISTED_PRODUCT_SPACE' IN types1 THEN
    n := SIZEOF (sp1\listed_product_space.factors);
    IF 'UNIFORM_PRODUCT_SPACE' IN types2 THEN
      IF n <> sp2\uniform_product_space.exponent THEN
        RETURN (FALSE);
      END_IF;
      REPEAT i := 1 TO n;
        IF NOT compatible_spaces(sp2\uniform_product_space.base,
          sp1\listed_product_space.factors[i]) THEN
          RETURN (FALSE);
        END_IF;
      END_REPEAT;
      RETURN (TRUE);
    END_IF;
    IF 'LISTED_PRODUCT_SPACE' IN types2 THEN
      IF n <> SIZEOF (sp2\listed_product_space.factors) THEN
        RETURN (FALSE);
      END_IF;
      REPEAT i := 1 TO n;
        IF NOT compatible_spaces(sp1\listed_product_space.factors[i],
          sp2\listed_product_space.factors[i]) THEN
          RETURN (FALSE);
        END_IF;
      END_REPEAT;
      RETURN (TRUE);
    END_IF;
    IF 'EXTENDED_TUPLE_SPACE' IN types2 THEN
```

© ISO 2002 — All rights reserved

```
      m := space_dimension(sp2\extended_tuple_space.base);
      IF n < m THEN
        RETURN (FALSE);
      END_IF;
      IF n = m THEN
        RETURN (compatible_spaces(sp1, sp2\extended_tuple_space.base));
      END_IF;
      RETURN (compatible_spaces(sp1, assoc_product_space(
        sp2\extended_tuple_space.base, make_uniform_product_space(
        sp2\extended_tuple_space.extender, n - m))));
    END_IF;
    IF (schema_prefix + 'FUNCTION_SPACE') IN types2 THEN
      RETURN (FALSE);
    END_IF;
    -- Should be unreachable.
    RETURN (TRUE);
  END_IF;
  IF 'EXTENDED_TUPLE_SPACE' IN types1 THEN
    IF ('UNIFORM_PRODUCT_SPACE' IN types2) OR
      ('LISTED_PRODUCT_SPACE' IN types2) THEN
      RETURN (compatible_spaces(sp2, sp1));
    END_IF;
    IF 'EXTENDED_TUPLE_SPACE' IN types2 THEN
      IF NOT compatible_spaces(sp1\extended_tuple_space.extender,
        sp2\extended_tuple_space.extender) THEN
        RETURN (FALSE);
      END_IF;
      n := space_dimension(sp1\extended_tuple_space.base);
      m := space_dimension(sp2\extended_tuple_space.base);
      IF n < m THEN
        RETURN (compatible_spaces(assoc_product_space(sp1\extended_tuple_space.base,
          make_uniform_product_space(sp1\extended_tuple_space.extender, m - n)),
          sp2\extended_tuple_space.base));
      END_IF;
      IF n = m THEN
        RETURN (compatible_spaces(sp1\extended_tuple_space.base,
          sp2\extended_tuple_space.base));
      END_IF;
      IF n > m THEN
        RETURN (compatible_spaces(sp1\extended_tuple_space.base,
          assoc_product_space(sp2\extended_tuple_space.base,
          make_uniform_product_space(sp2\extended_tuple_space.extender, n - m))));
      END_IF;
    END_IF;
    IF 'FUNCTION_SPACE' IN types2 THEN
      RETURN (FALSE);
    END_IF;
    -- Should be unreachable.
    RETURN (TRUE);
  END_IF;
  IF 'FUNCTION_SPACE' IN types1 THEN
    IF 'FUNCTION_SPACE' IN types2 THEN
      s1 := sp1\function_space.domain_argument;
      s2 := sp2\function_space.domain_argument;
      CASE sp1\function_space.domain_constraint OF
      sc_equal : BEGIN
        CASE sp2\function_space.domain_constraint OF
        sc_equal : lgcl := subspace_of(s1, s2) AND subspace_of(s2, s1);
        sc_subspace : lgcl := subspace_of(s1, s2);
```

 © ISO 2002 — All rights reserved

```
        sc_member : lgcl := member_of(s1, s2);
        END_CASE;
        END;
      sc_subspace :BEGIN
        CASE sp2\function_space.domain_constraint OF
        sc_equal : lgcl := subspace_of(s2, s1);
        sc_subspace : lgcl := compatible_spaces(s1, s2);
        sc_member : lgcl := UNKNOWN;
        END_CASE;
        END;
      sc_member :BEGIN
        CASE sp2\function_space.domain_constraint OF
        sc_equal : lgcl := member_of(s2, s1);
        sc_subspace : lgcl := UNKNOWN;
        sc_member : lgcl := compatible_spaces(s1, s2);
        END_CASE;
        END;
      END_CASE;
      IF lgcl = FALSE THEN
        RETURN (FALSE);
      END_IF;
      s1 := sp1\function_space.range_argument;
      s2 := sp2\function_space.range_argument;
      CASE sp1\function_space.range_constraint OF
      sc_equal : BEGIN
        CASE sp2\function_space.range_constraint OF
        sc_equal : lgcl := subspace_of(s1, s2) AND subspace_of(s2, s1);
        sc_subspace : lgcl := subspace_of(s1, s2);
        sc_member : lgcl := member_of(s1, s2);
        END_CASE;
        END;
      sc_subspace :BEGIN
        CASE sp2\function_space.range_constraint OF
        sc_equal : lgcl := subspace_of(s2, s1);
        sc_subspace : lgcl := compatible_spaces(s1, s2);
        sc_member : lgcl := UNKNOWN;
        END_CASE;
        END;
      sc_member :BEGIN
        CASE sp2\function_space.range_constraint OF
        sc_equal : lgcl := member_of(s2, s1);
        sc_subspace : lgcl := UNKNOWN;
        sc_member : lgcl := compatible_spaces(s1, s2);
        END_CASE;
        END;
      END_CASE;
      IF lgcl = FALSE THEN
        RETURN (FALSE);
      END_IF;
      RETURN (TRUE);
    END_IF;
    -- Should be unreachable.
    RETURN (TRUE);
  END_IF;
  -- Should be unreachable.
  RETURN (TRUE);
END_FUNCTION;  -- compatible_spaces
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

sp1: (input) The first space.

sp2: (input) The second space.

return: (output) A BOOLEAN value of FALSE is returned if it is determined that the intersection of the two spaces is empty. Otherwise, TRUE is returned.

4.6.16 composable_sequence

This function checks the suitability of forming a functional composition of a list of mathematical functions by verifying that the range of each one other than the last is compatible with the domain of the next one.

NOTE Two spaces are compatible if their intersection is not known to be empty. It does no harm to the schema if this test fails to detect some cases where the intersection is, in fact, empty. Such a function will have an actual domain which is empty, which will make it quite useless for any practical purpose. Erring in the other direction would prevent the exchange of functions which might have a practical use.

EXPRESS specification:

```
*)
FUNCTION composable_sequence(operands : LIST [2:?] OF maths_function) : BOOLEAN;
  REPEAT i := 1 TO SIZEOF (operands) - 1;
    IF NOT compatible_spaces (operands[i].range, operands[i+1].domain) THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- composable_sequence
(*
```

Argument definitions:

operands: (input) The list of **maths_function** instances to be checked for composability.

return: (output) A BOOLEAN value signifying whether the list is composable.

4.6.17 convert_to_literal

This function constructs an instance of **generic_literal** given a value of **maths_atom**.

EXPRESS specification:

```
*)
FUNCTION convert_to_literal(val : maths_atom) : generic_literal;
  LOCAL
    types : SET OF STRING := TYPEOF (val);
  END_LOCAL;
  IF 'INTEGER' IN types THEN  RETURN (make_int_literal (val));      END_IF;
  IF 'REAL'    IN types THEN  RETURN (make_real_literal (val));     END_IF;
```

 © ISO 2002 — All rights reserved

```
  IF 'BOOLEAN' IN types THEN  RETURN (make_boolean_literal (val));  END_IF;
  IF 'STRING'  IN types THEN  RETURN (make_string_literal (val));   END_IF;
  IF 'LOGICAL' IN types THEN  RETURN (make_logical_literal (val));  END_IF;
  IF 'BINARY'  IN types THEN  RETURN (make_binary_literal (val));   END_IF;
  IF (schema_prefix + 'MATHS_ENUM_ATOM') IN types THEN
    RETURN (make_maths_enum_literal (val));
  END_IF;
  -- Should be unreachable
  RETURN (?);
END_FUNCTION;  -- convert_to_literal
(*
```

Argument definitions:

val: (input) The **maths_atom** value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of some subtype of **generic_literal** representing the input value.

4.6.18 convert_to_maths_function

This function constructs a **maths_function** instance for any value of type **maths_function_-select** which isn't already of that type.

EXPRESS specification:

```
*)
FUNCTION convert_to_maths_function(func : maths_function_select) : maths_function;
  LOCAL
    efenum : elementary_function_enumerators;
    mthfun : maths_function;
  END_LOCAL;
  IF (schema_prefix + 'MATHS_FUNCTION') IN TYPEOF (func) THEN
    mthfun := func;
  ELSE
    efenum := func;
    mthfun := make_elementary_function (efenum);
  END_IF;
  RETURN (mthfun);
END_FUNCTION;  -- convert_to_maths_function
(*
```

Argument definitions:

func: (input) The value of type **maths_function_select** to be converted.

return: (output) An equivalent instance of **maths_function**.

4.6.19 convert_to_maths_value

This function returns a **maths_value** equivalent to any generic value which can be represented as a **maths_value**, and the indeterminate value, otherwise.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION convert_to_maths_value(val : GENERIC:G) : maths_value;
  LOCAL
    types : SET OF STRING := TYPEOF (val);
    ival  : maths_integer;
    rval  : maths_real;
    nval  : maths_number;
    tfval : maths_boolean;
    lval  : maths_logical;
    sval  : maths_string;
    bval  : maths_binary;
    tval  : maths_tuple := the_empty_maths_tuple;
    mval  : maths_value;
  END_LOCAL;
  IF (schema_prefix + 'MATHS_VALUE') IN types THEN  RETURN (val);  END_IF;
  IF 'INTEGER' IN types THEN  ival := val;   RETURN (ival);   END_IF;
  IF 'REAL'    IN types THEN  rval := val;   RETURN (rval);   END_IF;
  IF 'NUMBER'  IN types THEN  nval := val;   RETURN (nval);   END_IF;
  IF 'BOOLEAN' IN types THEN  tfval := val;  RETURN (tfval);  END_IF;
  IF 'LOGICAL' IN types THEN  lval := val;   RETURN (lval);   END_IF;
  IF 'STRING'  IN types THEN  sval := val;   RETURN (sval);   END_IF;
  IF 'BINARY'  IN types THEN  bval := val;   RETURN (bval);   END_IF;
  IF 'LIST' IN types THEN
    REPEAT i := 1 TO SIZEOF (val);
      mval := convert_to_maths_value (val[i]);
      IF NOT EXISTS (mval) THEN  RETURN (?);  END_IF;
      INSERT (tval, mval, i-1);
    END_REPEAT;
    RETURN (tval);
  END_IF;
  RETURN (?);
END_FUNCTION;  -- convert_to_maths_value
(*
```

Argument definitions:

val: (input) The arbitrary value to be converted.

return: (output) An equivalent value of type **maths_value**.

4.6.20 convert_to_operand

This function creates a generic expression corresponding to an arbitrary **maths_value**. This amounts to creating instances of **generic_literal** for those values which are not already instances of **generic_expression**.

EXPRESS specification:

```
*)
FUNCTION convert_to_operand(val : maths_value) : generic_expression;
  LOCAL
    types  : SET OF STRING := stripped_typeof (val);
```

 © ISO 2002 — All rights reserved

```
  END_LOCAL;
  -- Use intermediate variables of appropriate declared types to help the compilers.
  IF 'GENERIC_EXPRESSION' IN types THEN  RETURN (val);  END_IF;
  IF 'MATHS_ATOM' IN types THEN  RETURN (convert_to_literal (val));  END_IF;
  IF 'ATOM_BASED_VALUE' IN types THEN  RETURN (make_atom_based_literal(val));  END_IF;
  IF 'MATHS_TUPLE' IN types THEN  RETURN (make_maths_tuple_literal(val));  END_IF;
  -- Should be unreachable
  RETURN (?);
END_FUNCTION;  -- convert_to_operand
(*
```

Argument definitions:

val: (input) The arbitrary value which is to be converted into a generic expression.

return: (output) The generic expression corresponding to the input.

4.6.21 convert_to_operands

This function creates a list of generic expressions corresponding to an arbitrary aggregate of **maths_value**. The formation of subsidiary **maths_tuple** values is translated into function applications of the tuple-forming elementary function, the values which are elementary EXPRESS values are translated into appropriate literals, and those values which are already generic expressions are translated into themselves. The order of the operands in the list corresponds to the indexing order of the input aggregate.

EXPRESS specification:

```
*)
FUNCTION convert_to_operands(values : AGGREGATE OF maths_value)
                            : LIST OF generic_expression;
  LOCAL
    operands : LIST OF generic_expression := [];
    loc : INTEGER := 0;
  END_LOCAL;
  IF NOT EXISTS (values) THEN  RETURN (?);  END_IF;
  REPEAT i := LOINDEX (values) TO HIINDEX (values);
    INSERT (operands, convert_to_operand (values[i]), loc);
    loc := loc + 1;
  END_REPEAT;
  RETURN (operands);
END_FUNCTION;  -- convert_to_operands
(*
```

Argument definitions:

values: (input) The aggregate of arbitrary values which are to be converted into a list of generic expressions.

return: (output) The list of generic expressions corresponding to the input.

© ISO 2002 — All rights reserved

4.6.22 convert_to_operands_prcmfn

This function constructs a value for the inherited **operands** attribute of an instance of **parallel_composed_function** from its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION convert_to_operands_prcmfn(srcdom  : maths_space_or_function;
                                    prepfun : LIST OF maths_function;
                                    finfun  : maths_function_select)
                                   : LIST [2:?] OF generic_expression;
  LOCAL
    operands : LIST OF generic_expression := [];
  END_LOCAL;
  INSERT (operands, srcdom, 0);
  REPEAT i := 1 TO SIZEOF (prepfun);
    INSERT (operands, prepfun[i], i);
  END_REPEAT;
  INSERT (operands, convert_to_maths_function (finfun), SIZEOF (prepfun)+1);
  RETURN (operands);
END_FUNCTION;  -- convert_to_operands_prcmfn
(*
```

Argument definitions:

srcdom: (input) The value of the **source_of_domain** attribute.

prepfun: (input) The value of the **prep_functions** attribute.

finfun: (input) The value of the **final_function** attribute.

return: (output) The value to be used for the inherited **operands** attribute.

4.6.23 definite_integral_check

This function verifies whether or not its arguments are consistent with certain requirements for the existence of a mathematical integral. Specifically, it verifies that the value identifying the implicit variable of integration is within range, that the corresponding space of possible input values is a real interval, that, if the lower limit of integration is negative infinity, then the real interval is unbounded below, and that, if the upper limit of integration is positive infinity, then the real interval is unbounded above.

EXPRESS specification:

```
*)
FUNCTION definite_integral_check(domain   : tuple_space;
                                 vrblint  : input_selector;
                                 lowerinf : BOOLEAN;
                                 upperinf : BOOLEAN) : BOOLEAN;
  LOCAL
    domn : tuple_space := domain;
    fspc : maths_space;
```

 © ISO 2002 — All rights reserved

```
    dim : nonnegative_integer;
    k : positive_integer;
  END_LOCAL;
  IF (space_dimension (domain) = 1) AND ((schema_prefix + 'TUPLE_SPACE') IN
    TYPEOF (factor1 (domain))) THEN
    domn := factor1 (domain);
  END_IF;
  dim := space_dimension (domn);
  k := vrblint;
  IF k > dim THEN  RETURN (FALSE);  END_IF;
  fspc := factor_space (domn, k);
  IF NOT ((schema_prefix + 'REAL_INTERVAL') IN TYPEOF (fspc)) THEN
    RETURN (FALSE);
  END_IF;
  IF lowerinf AND min_exists (fspc) THEN  RETURN (FALSE);  END_IF;
  IF upperinf AND max_exists (fspc) THEN  RETURN (FALSE);  END_IF;
  RETURN (TRUE);
END_FUNCTION;  -- definite_integral_check
(*
```

Argument definitions:

domain: (input) The domain of the integrand function.

vrblint: (input) The index of the implicit variable of integration.

lowerinf: (input) Indicator for whether the lower limit of integration is negative infinity or a real number.

upperinf: (input) Indicator for whether the upper limit of integration is positive infinity or a real number.

return: (output) A BOOLEAN value indicating whether or not the four arguments are consistent with the existence of a mathematical integral.

4.6.24 definite_integral_expr_check

This function verifies whether or not its arguments are consistent with certain requirements for the existence of a mathematical integral. Specifically, it verifies that the number of operands is two plus one for the lower limit of integration if **lowerinf** is FALSE plus one for the upper limit of integration if **upperinf** is FALSE, that the second operand is a variable whose values space is a real interval, that the real interval is unbounded below if **lowerinf** is TRUE, that the real interval is unbounded above if **upperinf** is TRUE, and that the values spaces of the third and fourth operands, when present, are compatible with the real interval.

EXPRESS specification:

```
*)
FUNCTION definite_integral_expr_check(operands : LIST [2:?] OF generic_expression;
                                      lowerinf : BOOLEAN;
                                      upperinf : BOOLEAN) : BOOLEAN;
  LOCAL
    nops : INTEGER := 2;
    vspc : maths_space;
    dim : nonnegative_integer;
```

© ISO 2002 — All rights reserved

```
    k : positive_integer;
    bspc : maths_space;
  END_LOCAL;
  IF NOT lowerinf THEN  nops := nops + 1;  END_IF;
  IF NOT upperinf THEN  nops := nops + 1;  END_IF;
  IF SIZEOF (operands) <> nops THEN  RETURN (FALSE);  END_IF;
  IF NOT ('GENERIC_VARIABLE' IN stripped_typeof(operands[2])) THEN
    RETURN (FALSE);
  END_IF;
  IF NOT has_values_space (operands[2]) THEN  RETURN (FALSE);  END_IF;
  vspc := values_space_of (operands[2]);
  IF NOT ('REAL_INTERVAL' IN stripped_typeof(vspc)) THEN  RETURN (FALSE);  END_IF;
  IF lowerinf THEN
    IF min_exists (vspc) THEN  RETURN (FALSE);  END_IF;
    k := 3;
  ELSE
    IF NOT has_values_space (operands[3]) THEN  RETURN (FALSE);  END_IF;
    bspc := values_space_of (operands[3]);
    IF NOT compatible_spaces (bspc, vspc) THEN  RETURN (FALSE);  END_IF;
    k := 4;
  END_IF;
  IF upperinf THEN
    IF max_exists (vspc) THEN  RETURN (FALSE);  END_IF;
  ELSE
    IF NOT has_values_space (operands[k]) THEN  RETURN (FALSE);  END_IF;
    bspc := values_space_of (operands[k]);
    IF NOT compatible_spaces (bspc, vspc) THEN  RETURN (FALSE);  END_IF;
  END_IF;
  RETURN (TRUE);
END_FUNCTION;  -- definite_integral_expr_check
(*
```

Argument definitions:

operands: (input) The list of operands for a proposed instance of **definite_integral_expression**.

lowerinf: (input) Indicator for whether the lower limit of integration is negative infinity or an operand.

upperinf: (input) Indicator for whether the upper limit of integration is positive infinity or an operand.

return: (output) A BOOLEAN value indicating whether or not the three arguments are consistent with the existence of a mathematical integral.

4.6.25 derive_definite_integral_domain

This function constructs a value for the inherited domain attribute of an instance of **definite_integral_function**. It prefixes zero, one or two copies of the domain of the variable of integration to the domain of the integrand minus the factor space corresponding to the variable of integration. The number of prefixed factors matches the number of finite bounds of the integral. The packaging is chosen to match that of the integrand function.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION derive_definite_integral_domain(igrl : definite_integral_function)
                                         : tuple_space;

  -- Internal utility function:
  FUNCTION process_product_space(spc         : product_space;
                                 idx, prefix : INTEGER;
                                 vdomn       : maths_space) : product_space;
    LOCAL
      uspc : uniform_product_space;
      expnt : INTEGER;
      factors : LIST OF maths_space;
    END_LOCAL;
    IF (schema_prefix + 'UNIFORM_PRODUCT_SPACE') IN TYPEOF (spc) THEN
      uspc := spc;
      expnt := uspc.exponent + prefix;
      IF idx <= uspc.exponent THEN  expnt := expnt - 1;  END_IF;
      IF expnt = 0 THEN
        RETURN (make_listed_product_space([]));
      ELSE
        RETURN (make_uniform_product_space(uspc.base,expnt));
      END_IF;
    ELSE
      factors := spc\listed_product_space.factors;
      IF idx <= SIZEOF (factors) THEN  REMOVE (factors, idx);  END_IF;
      IF prefix > 0 THEN
        INSERT (factors, vdomn, 0);
        IF prefix > 1 THEN  INSERT (factors, vdomn, 0);  END_IF;
      END_IF;
      RETURN (make_listed_product_space(factors));
    END_IF;
  END_FUNCTION;  -- process_product_space

  -- Resume body of derive_definite_integral_domain function
  LOCAL
    idomn : tuple_space := igrl.integrand.domain;
    types : SET OF STRING := TYPEOF (idomn);
    idx : INTEGER := igrl.variable_of_integration;
    tupled : BOOLEAN := bool(((space_dimension(idomn) = 1) AND
                         ((schema_prefix + 'TUPLE_SPACE') IN types)));
    prefix : INTEGER := 0;
    espc : extended_tuple_space;
    vdomn : maths_space;
  END_LOCAL;
  IF tupled THEN
    idomn := factor1(idomn);
    types := TYPEOF (idomn);
  END_IF;
  IF igrl.lower_limit_neg_infinity THEN  prefix := prefix + 1;  END_IF;
  IF igrl.upper_limit_pos_infinity THEN  prefix := prefix + 1;  END_IF;
  vdomn := factor_space(idomn,idx);
  IF (schema_prefix + 'EXTENDED_TUPLE_SPACE') IN types THEN
    espc := idomn;
    idomn := make_extended_tuple_space(process_product_space(espc.base,idx,
      prefix,vdomn),espc.extender);
  ELSE
    idomn := process_product_space(idomn,idx,prefix,vdomn);
```

© ISO 2002 — All rights reserved

```
  END_IF;
  IF tupled THEN  RETURN (one_tuples_of(idomn));
  ELSE            RETURN (idomn);                END_IF;
END_FUNCTION;  -- derive_definite_integral_domain
(*
```

Argument definitions:

igrl: (input) The instance of **definite_integral_function** whose domain is to be derived.

return: (output) The function domain of the input instance.

4.6.26 derive_elementary_function_domain

This function constructs a value for the inherited domain attribute of an instance of **elementary_function** from the corresponding **elementary_function_enumerators** enumeration item value.

EXPRESS specification:

```
*)
FUNCTION derive_elementary_function_domain(ef_val : elementary_function_enumerators)
                                        : tuple_space;
  IF NOT EXISTS (ef_val) THEN  RETURN (?);  END_IF;
  CASE ef_val OF
  ef_and : RETURN (make_extended_tuple_space (the_zero_tuple_space, the_logicals));
  ef_or : RETURN (make_extended_tuple_space (the_zero_tuple_space, the_logicals));
  ef_not : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_xor : RETURN (make_uniform_product_space (the_logicals, 2));
  ef_negate_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_add_i : RETURN (the_integer_tuples);
  ef_subtract_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_multiply_i : RETURN (the_integer_tuples);
  ef_divide_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_mod_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_exponentiate_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_eq_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_ne_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_gt_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_lt_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_ge_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_le_i : RETURN (make_uniform_product_space (the_integers, 2));
  ef_abs_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_if_i : RETURN (make_listed_product_space ([the_logicals, the_integers,
    the_integers]));
  ef_negate_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_reciprocal_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_add_r : RETURN (the_real_tuples);
  ef_subtract_r : RETURN (make_uniform_product_space (the_reals, 2));
  ef_multiply_r : RETURN (the_real_tuples);
  ef_divide_r : RETURN (make_uniform_product_space (the_reals, 2));
  ef_mod_r : RETURN (make_uniform_product_space (the_reals, 2));
  ef_exponentiate_r : RETURN (make_listed_product_space ([the_nonnegative_reals,
    the_reals]));
  ef_exponentiate_ri : RETURN (make_listed_product_space ([the_reals, the_integers]));
```

 © ISO 2002 — All rights reserved

```
ef_eq_r : RETURN (make_uniform_product_space (the_reals, 2));
ef_ne_r : RETURN (make_uniform_product_space (the_reals, 2));
ef_gt_r : RETURN (make_uniform_product_space (the_reals, 2));
ef_lt_r : RETURN (make_uniform_product_space (the_reals, 2));
ef_ge_r : RETURN (make_uniform_product_space (the_reals, 2));
ef_le_r : RETURN (make_uniform_product_space (the_reals, 2));
ef_abs_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_acos_r : RETURN (make_uniform_product_space (the_neg1_one_interval, 1));
ef_asin_r : RETURN (make_uniform_product_space (the_neg1_one_interval, 1));
ef_atan2_r : RETURN (make_uniform_product_space (the_reals, 2));
ef_cos_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_exp_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_ln_r : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
ef_log2_r : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
ef_log10_r : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
ef_sin_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_sqrt_r : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
ef_tan_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_if_r : RETURN (make_listed_product_space ([the_logicals, the_reals, the_reals]));
ef_negate_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_reciprocal_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_add_c : RETURN (the_complex_tuples);
ef_subtract_c : RETURN (make_uniform_product_space (the_complex_numbers, 2));
ef_multiply_c : RETURN (the_complex_tuples);
ef_divide_c : RETURN (make_uniform_product_space (the_complex_numbers, 2));
ef_exponentiate_c : RETURN (make_uniform_product_space (the_complex_numbers, 2));
ef_exponentiate_ci : RETURN (make_listed_product_space ([the_complex_numbers,
  the_integers]));
ef_eq_c : RETURN (make_uniform_product_space (the_complex_numbers, 2));
ef_ne_c : RETURN (make_uniform_product_space (the_complex_numbers, 2));
ef_conjugate_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_abs_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_arg_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_cos_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_exp_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_ln_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_sin_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_sqrt_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_tan_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_if_c : RETURN (make_listed_product_space ([the_logicals, the_complex_numbers,
  the_complex_numbers]));
ef_subscript_s : RETURN (make_listed_product_space ([the_strings, the_integers]));
ef_eq_s : RETURN (make_uniform_product_space (the_strings, 2));
ef_ne_s : RETURN (make_uniform_product_space (the_strings, 2));
ef_gt_s : RETURN (make_uniform_product_space (the_strings, 2));
ef_lt_s : RETURN (make_uniform_product_space (the_strings, 2));
ef_ge_s : RETURN (make_uniform_product_space (the_strings, 2));
ef_le_s : RETURN (make_uniform_product_space (the_strings, 2));
ef_subsequence_s : RETURN (make_listed_product_space ([the_strings, the_integers,
  the_integers]));
ef_concat_s : RETURN (make_extended_tuple_space (the_zero_tuple_space, the_strings));
ef_size_s : RETURN (make_uniform_product_space (the_strings, 1));
ef_format : RETURN (make_listed_product_space ([the_numbers, the_strings]));
ef_value : RETURN (make_uniform_product_space (the_strings, 1));
ef_like : RETURN (make_uniform_product_space (the_strings, 2));
ef_if_s : RETURN (make_listed_product_space ([the_logicals, the_strings,
  the_strings]));
ef_subscript_b : RETURN (make_listed_product_space ([the_binarys, the_integers]));
ef_eq_b : RETURN (make_uniform_product_space (the_binarys, 2));
```

© ISO 2002 — All rights reserved

```
  ef_ne_b : RETURN (make_uniform_product_space (the_binarys, 2));
  ef_gt_b : RETURN (make_uniform_product_space (the_binarys, 2));
  ef_lt_b : RETURN (make_uniform_product_space (the_binarys, 2));
  ef_ge_b : RETURN (make_uniform_product_space (the_binarys, 2));
  ef_le_b : RETURN (make_uniform_product_space (the_binarys, 2));
  ef_subsequence_b : RETURN (make_listed_product_space ([the_binarys, the_integers,
    the_integers]));
  ef_concat_b : RETURN (make_extended_tuple_space (the_zero_tuple_space, the_binarys));
  ef_size_b : RETURN (make_uniform_product_space (the_binarys, 1));
  ef_if_b : RETURN (make_listed_product_space ([the_logicals, the_binarys,
    the_binarys]));
  ef_subscript_t : RETURN (make_listed_product_space ([the_tuples, the_integers]));
  ef_eq_t : RETURN (make_uniform_product_space (the_tuples, 2));
  ef_ne_t : RETURN (make_uniform_product_space (the_tuples, 2));
  ef_concat_t : RETURN (make_extended_tuple_space (the_zero_tuple_space, the_tuples));
  ef_size_t : RETURN (make_uniform_product_space (the_tuples, 1));
  ef_entuple : RETURN (the_tuples);
  ef_detuple : RETURN (make_uniform_product_space (the_generics, 1));
  ef_insert : RETURN (make_listed_product_space ([the_tuples, the_generics,
    the_integers]));
  ef_remove : RETURN (make_listed_product_space ([the_tuples, the_integers]));
  ef_if_t : RETURN (make_listed_product_space ([the_logicals, the_tuples,
    the_tuples]));
  ef_sum_it : RETURN (make_uniform_product_space (the_integer_tuples, 1));
  ef_product_it : RETURN (make_uniform_product_space (the_integer_tuples, 1));
  ef_add_it : RETURN (make_extended_tuple_space (the_integer_tuples,
    the_integer_tuples));
  ef_subtract_it : RETURN (make_uniform_product_space (the_integer_tuples, 2));
  ef_scalar_mult_it : RETURN (make_listed_product_space ([the_integers,
    the_integer_tuples]));
  ef_dot_prod_it : RETURN (make_uniform_product_space (the_integer_tuples, 2));
  ef_sum_rt : RETURN (make_uniform_product_space (the_real_tuples, 1));
  ef_product_rt : RETURN (make_uniform_product_space (the_real_tuples, 1));
  ef_add_rt : RETURN (make_extended_tuple_space (the_real_tuples, the_real_tuples));
  ef_subtract_rt : RETURN (make_uniform_product_space (the_real_tuples, 2));
  ef_scalar_mult_rt : RETURN (make_listed_product_space ([the_reals,
    the_real_tuples]));
  ef_dot_prod_rt : RETURN (make_uniform_product_space (the_real_tuples, 2));
  ef_norm_rt : RETURN (make_uniform_product_space (the_real_tuples, 1));
  ef_sum_ct : RETURN (make_uniform_product_space (the_complex_tuples, 1));
  ef_product_ct : RETURN (make_uniform_product_space (the_complex_tuples, 1));
  ef_add_ct : RETURN (make_extended_tuple_space (the_complex_tuples,
    the_complex_tuples));
  ef_subtract_ct : RETURN (make_uniform_product_space (the_complex_tuples, 2));
  ef_scalar_mult_ct : RETURN (make_listed_product_space ([the_complex_numbers,
    the_complex_tuples]));
  ef_dot_prod_ct : RETURN (make_uniform_product_space (the_complex_tuples, 2));
  ef_norm_ct : RETURN (make_uniform_product_space (the_complex_tuples, 1));
  ef_if : RETURN (make_listed_product_space ([the_logicals, the_generics,
    the_generics]));
  ef_ensemble : RETURN (the_tuples);
  ef_member_of : RETURN (make_listed_product_space ([the_generics, the_maths_spaces]));
  OTHERWISE : RETURN (?);
  END_CASE;
END_FUNCTION;  -- derive_elementary_function_domain
(*
```

 © ISO 2002 — All rights reserved

Argument definitions:

ef_val: (input) The enumeration value from **elementary_function_enumerators** identifying the elementary function whose comain space is to be constructed.

return: (output) The constructed domain space.

4.6.27 derive_elementary_function_range

This function constructs a value for the inherited range attribute of an instance of **elementary_-function** from the corresponding **elementary_function_enumerators** enumeration item value.

EXPRESS specification:

```
*)
FUNCTION derive_elementary_function_range(ef_val : elementary_function_enumerators)
                                          : tuple_space;
  IF NOT EXISTS (ef_val) THEN  RETURN (?);  END_IF;
  CASE ef_val OF
  ef_and : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_or : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_not : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_xor : RETURN (make_uniform_product_space (the_logicals, 2));
  ef_negate_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_add_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_subtract_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_multiply_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_divide_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_mod_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_exponentiate_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_eq_i : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_ne_i : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_gt_i : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_lt_i : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_ge_i : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_le_i : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_abs_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_if_i : RETURN (make_uniform_product_space (the_integers, 1));
  ef_negate_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_reciprocal_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_add_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_subtract_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_multiply_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_divide_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_mod_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_exponentiate_r : RETURN (make_uniform_product_space (the_reals, 1));
  ef_exponentiate_ri : RETURN (make_uniform_product_space (the_reals, 1));
  ef_eq_r : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_ne_r : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_gt_r : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_lt_r : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_ge_r : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_le_r : RETURN (make_uniform_product_space (the_logicals, 1));
  ef_abs_r : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
  ef_acos_r : RETURN (make_uniform_product_space (the_zero_pi_interval, 1));
```

© ISO 2002 — All rights reserved

```
ef_asin_r : RETURN (make_uniform_product_space (the_neghalfpi_halfpi_interval, 1));
ef_atan2_r : RETURN (make_uniform_product_space (the_negpi_pi_interval, 1));
ef_cos_r : RETURN (make_uniform_product_space (the_neg1_one_interval, 1));
ef_exp_r : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
ef_ln_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_log2_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_log10_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_sin_r : RETURN (make_uniform_product_space (the_neg1_one_interval, 1));
ef_sqrt_r : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
ef_tan_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_if_r : RETURN (make_uniform_product_space (the_reals, 1));
ef_negate_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_reciprocal_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_add_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_subtract_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_multiply_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_divide_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_exponentiate_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_exponentiate_ci : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_eq_c : RETURN (make_uniform_product_space (the_logicals, 1));
ef_ne_c : RETURN (make_uniform_product_space (the_logicals, 1));
ef_conjugate_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_abs_c : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
ef_arg_c : RETURN (make_uniform_product_space (the_negpi_pi_interval, 1));
ef_cos_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_exp_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_ln_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_sin_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_sqrt_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_tan_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_if_c : RETURN (make_uniform_product_space (the_complex_numbers, 1));
ef_subscript_s : RETURN (make_uniform_product_space (the_strings, 1));
ef_eq_s : RETURN (make_uniform_product_space (the_logicals, 1));
ef_ne_s : RETURN (make_uniform_product_space (the_logicals, 1));
ef_gt_s : RETURN (make_uniform_product_space (the_logicals, 1));
ef_lt_s : RETURN (make_uniform_product_space (the_logicals, 1));
ef_ge_s : RETURN (make_uniform_product_space (the_logicals, 1));
ef_le_s : RETURN (make_uniform_product_space (the_logicals, 1));
ef_subsequence_s : RETURN (make_uniform_product_space (the_strings, 1));
ef_concat_s : RETURN (make_uniform_product_space (the_strings, 1));
ef_size_s : RETURN (make_uniform_product_space (the_integers, 1));
ef_format : RETURN (make_uniform_product_space (the_strings, 1));
ef_value : RETURN (make_uniform_product_space (the_reals, 1));
ef_like : RETURN (make_uniform_product_space (the_booleans, 1));
ef_if_s : RETURN (make_uniform_product_space (the_strings, 1));
ef_subscript_b : RETURN (make_uniform_product_space (the_binarys, 1));
ef_eq_b : RETURN (make_uniform_product_space (the_logicals, 1));
ef_ne_b : RETURN (make_uniform_product_space (the_logicals, 1));
ef_gt_b : RETURN (make_uniform_product_space (the_logicals, 1));
ef_lt_b : RETURN (make_uniform_product_space (the_logicals, 1));
ef_ge_b : RETURN (make_uniform_product_space (the_logicals, 1));
ef_le_b : RETURN (make_uniform_product_space (the_logicals, 1));
ef_subsequence_b : RETURN (make_uniform_product_space (the_binarys, 1));
ef_concat_b : RETURN (make_uniform_product_space (the_binarys, 1));
ef_size_b : RETURN (make_uniform_product_space (the_integers, 1));
ef_if_b : RETURN (make_uniform_product_space (the_binarys, 1));
ef_subscript_t : RETURN (make_uniform_product_space (the_generics, 1));
ef_eq_t : RETURN (make_uniform_product_space (the_logicals, 1));
ef_ne_t : RETURN (make_uniform_product_space (the_logicals, 1));
```

 © ISO 2002 — All rights reserved

```
  ef_concat_t : RETURN (make_uniform_product_space (the_tuples, 1));
  ef_size_t : RETURN (make_uniform_product_space (the_integers, 1));
  ef_entuple : RETURN (make_uniform_product_space (the_tuples, 1));
  ef_detuple : RETURN (the_tuples);
  ef_insert : RETURN (make_uniform_product_space (the_tuples, 1));
  ef_remove : RETURN (make_uniform_product_space (the_tuples, 1));
  ef_if_t : RETURN (make_uniform_product_space (the_tuples, 1));
  ef_sum_it : RETURN (make_uniform_product_space (the_integers, 1));
  ef_product_it : RETURN (make_uniform_product_space (the_integers, 1));
  ef_add_it : RETURN (make_uniform_product_space (the_integer_tuples, 1));
  ef_subtract_it : RETURN (make_uniform_product_space (the_integer_tuples, 1));
  ef_scalar_mult_it : RETURN (make_uniform_product_space (the_integer_tuples, 1));
  ef_dot_prod_it : RETURN (make_uniform_product_space (the_integers, 1));
  ef_sum_rt : RETURN (make_uniform_product_space (the_reals, 1));
  ef_product_rt : RETURN (make_uniform_product_space (the_reals, 1));
  ef_add_rt : RETURN (make_uniform_product_space (the_real_tuples, 1));
  ef_subtract_rt : RETURN (make_uniform_product_space (the_real_tuples, 1));
  ef_scalar_mult_rt : RETURN (make_uniform_product_space (the_real_tuples, 1));
  ef_dot_prod_rt : RETURN (make_uniform_product_space (the_reals, 1));
  ef_norm_rt : RETURN (make_uniform_product_space (the_reals, 1));
  ef_sum_ct : RETURN (make_uniform_product_space (the_complex_numbers, 1));
  ef_product_ct : RETURN (make_uniform_product_space (the_complex_numbers, 1));
  ef_add_ct : RETURN (make_uniform_product_space (the_complex_tuples, 1));
  ef_subtract_ct : RETURN (make_uniform_product_space (the_complex_tuples, 1));
  ef_scalar_mult_ct : RETURN (make_uniform_product_space (the_complex_tuples, 1));
  ef_dot_prod_ct : RETURN (make_uniform_product_space (the_complex_numbers, 1));
  ef_norm_ct : RETURN (make_uniform_product_space (the_nonnegative_reals, 1));
  ef_if : RETURN (make_uniform_product_space (the_generics, 1));
  ef_ensemble : RETURN (make_uniform_product_space (the_maths_spaces, 1));
  ef_member_of : RETURN (make_uniform_product_space (the_logicals, 1));
  OTHERWISE : RETURN (?);
  END_CASE;
END_FUNCTION;  -- derive_elementary_function_range
(*
```

Argument definitions:

ef_val: (input) The enumeration value from **elementary_function_enumerators** identifying the elementary function whose range space is to be constructed.

return: (output) The constructed range space.

4.6.28 derive_finite_function_domain

This function returns an instance of **tuple_space** representing the domain of the **finite_function** to whose **pairs** attribute it is applied.

EXPRESS specification:

```
*)
FUNCTION derive_finite_function_domain(pairs : SET [1:?] OF LIST [2:2] OF maths_value)
                                       : tuple_space;
  LOCAL
    result : SET OF maths_value := [];
  END_LOCAL;
```

© ISO 2002 — All rights reserved

```
-- An ambiguity in ISO 10303-11:1994 pages 99-101 leaves the result of the following
-- three lines ambiguous in those cases where an operand is simultaneously a member
-- of the base type and the aggregate type.
-- REPEAT i := 1 TO SIZEOF (pairs);
--   result := result + pairs[i][1];
-- END_REPEAT;
-- The next line unions an empty set and the desired list to get the desired set.
  result := result + list_selected_components (pairs, 1);
  RETURN (one_tuples_of (make_finite_space (result)));
END_FUNCTION;  -- derive_finite_function_domain
(*
```

Argument definitions:

pairs: (input) The set of ordered pairs defining the function.

return: (output) An instance of **tuple_space** representing the domain of the function.

4.6.29 derive_finite_function_range

This function returns an instance of **tuple_space** representing the range of the **finite_function** to whose **pairs** attribute it is applied.

EXPRESS specification:

```
*)
FUNCTION derive_finite_function_range(pairs : SET [1:?] OF LIST [2:2] OF maths_value)
                                     : tuple_space;
  LOCAL
    result : SET OF maths_value := [];
  END_LOCAL;
-- An ambiguity in ISO 10303-11:1994 pages 99-101 leaves the result of the following
-- three lines ambiguous in those cases where an operand is simultaneously a member
-- of the base type and the aggregate type.
-- REPEAT i := 1 TO SIZEOF (pairs);
--   result := result + pairs[i][2];
-- END_REPEAT;
-- The next line unions an empty set and the desired list to get the desired set.
  result := result + list_selected_components (pairs, 2);
  RETURN (one_tuples_of (make_finite_space (result)));
END_FUNCTION;  -- derive_finite_function_range
(*
```

Argument definitions:

pairs: (input) The set of ordered pairs defining the function.

return: (output) An instance of **tuple_space** representing the range of the function.

4.6.30 derive_function_domain

This function returns an instance of **tuple_space** representing the domain of the **maths_-function** to which it is applied.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION derive_function_domain(func : maths_function) : tuple_space;
  LOCAL
    typenames : SET OF STRING := stripped_typeof(func);
    tspace : tuple_space := make_listed_product_space ([]);
    shape : LIST OF positive_integer;
    sidxs  : LIST OF INTEGER := [0];
    itvl   : finite_integer_interval;
    factors : LIST OF finite_integer_interval := [];
    is_uniform : BOOLEAN := TRUE;
  END_LOCAL;
  IF 'FINITE_FUNCTION' IN typenames THEN
    RETURN (derive_finite_function_domain (func\finite_function.pairs));
  END_IF;
  IF 'CONSTANT_FUNCTION' IN typenames THEN
    RETURN (domain_from (func\constant_function.source_of_domain));
  END_IF;
  IF 'SELECTOR_FUNCTION' IN typenames THEN
    RETURN (domain_from (func\selector_function.source_of_domain));
  END_IF;
  IF 'ELEMENTARY_FUNCTION' IN typenames THEN
    RETURN (derive_elementary_function_domain (func\elementary_function.func_id));
  END_IF;
  IF 'RESTRICTION_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (func\restriction_function.operand));
  END_IF;
  IF 'REPACKAGING_FUNCTION' IN typenames THEN
    IF func\repackaging_function.input_repack = ro_nochange THEN
      RETURN (func\repackaging_function.operand.domain);
    END_IF;
    IF func\repackaging_function.input_repack = ro_wrap_as_tuple THEN
      RETURN (factor1 (func\repackaging_function.operand.domain));
    END_IF;
    IF func\repackaging_function.input_repack = ro_unwrap_tuple THEN
      RETURN (one_tuples_of (func\repackaging_function.operand.domain));
    END_IF;
    -- Unreachable, as there is no other possible value for input_repack.
    RETURN (?);
  END_IF;
  IF 'REINDEXED_ARRAY_FUNCTION' IN typenames THEN
    shape := shape_of_array(func\unary_generic_expression.operand);
    sidxs := func\reindexed_array_function.starting_indices;
    REPEAT i := 1 TO SIZEOF (shape);
      itvl := make_finite_integer_interval (sidxs[i], sidxs[i]+shape[i]-1);
      INSERT (factors, itvl, i-1);
      IF shape[i] <> shape[1] THEN  is_uniform := FALSE;  END_IF;
    END_REPEAT;
    IF is_uniform THEN
      RETURN (make_uniform_product_space (factors[1], SIZEOF (shape)));
    END_IF;
    RETURN (make_listed_product_space (factors));
  END_IF;
  IF 'SERIES_COMPOSED_FUNCTION' IN typenames THEN
    RETURN (func\series_composed_function.operands[1].domain);
  END_IF;
  IF 'PARALLEL_COMPOSED_FUNCTION' IN typenames THEN
    RETURN (domain_from (func\parallel_composed_function.source_of_domain));
```

© ISO 2002 — All rights reserved

```
END_IF;
IF 'EXPLICIT_TABLE_FUNCTION' IN typenames THEN
  shape := func\explicit_table_function.shape;
  sidxs[1] := func\explicit_table_function.index_base;
  REPEAT i := 1 TO SIZEOF (shape);
    itvl := make_finite_integer_interval (sidxs[1], sidxs[1]+shape[i]-1);
    INSERT (factors, itvl, i-1);
    IF shape[i] <> shape[1] THEN  is_uniform := FALSE;  END_IF;
  END_REPEAT;
  IF is_uniform THEN
    RETURN (make_uniform_product_space (factors[1], SIZEOF (shape)));
  END_IF;
  RETURN (make_listed_product_space (factors));
END_IF;
IF 'HOMOGENEOUS_LINEAR_FUNCTION' IN typenames THEN
  RETURN (one_tuples_of (make_uniform_product_space
    (factor1 (func\homogeneous_linear_function.mat.range),
    func\homogeneous_linear_function.mat\explicit_table_function.shape
    [func\homogeneous_linear_function.sum_index])));
END_IF;
IF 'GENERAL_LINEAR_FUNCTION' IN typenames THEN
  RETURN (one_tuples_of (make_uniform_product_space
    (factor1 (func\general_linear_function.mat.range),
    func\general_linear_function.mat\explicit_table_function.shape
    [func\general_linear_function.sum_index] - 1)));
END_IF;
IF 'B_SPLINE_BASIS' IN typenames THEN
  RETURN (one_tuples_of (make_finite_real_interval
    (func\b_spline_basis.repeated_knots[func\b_spline_basis.order], closed,
    func\b_spline_basis.repeated_knots[func\b_spline_basis.num_basis+1], closed)));
END_IF;
IF 'B_SPLINE_FUNCTION' IN typenames THEN
  REPEAT i := 1 TO SIZEOF (func\b_spline_function.basis);
    tspace := assoc_product_space (tspace, func\b_spline_function.basis[i].domain);
  END_REPEAT;
  RETURN (one_tuples_of (tspace));
END_IF;
IF 'RATIONALIZE_FUNCTION' IN typenames THEN
  RETURN (func\rationalize_function.fun.domain);
END_IF;
IF 'PARTIAL_DERIVATIVE_FUNCTION' IN typenames THEN
  RETURN (func\partial_derivative_function.derivand.domain);
END_IF;
IF 'DEFINITE_INTEGRAL_FUNCTION' IN typenames THEN
  RETURN (derive_definite_integral_domain(func));
END_IF;
IF 'ABSTRACTED_EXPRESSION_FUNCTION' IN typenames THEN
  REPEAT i := 1 TO SIZEOF (func\abstracted_expression_function.variables);
    tspace := assoc_product_space (tspace, one_tuples_of (values_space_of
      (func\abstracted_expression_function.variables[i])));
  END_REPEAT;
  RETURN (tspace);
END_IF;
IF 'EXPRESSION_DENOTED_FUNCTION' IN typenames THEN
  RETURN (values_space_of (func\expression_denoted_function.expr)\function_space.
    domain_argument);
END_IF;
IF 'IMPORTED_POINT_FUNCTION' IN typenames THEN
  RETURN (one_tuples_of (make_listed_product_space ([])));
```

 © ISO 2002 — All rights reserved

```
  END_IF;
  IF 'IMPORTED_CURVE_FUNCTION' IN typenames THEN
    RETURN (func\imported_curve_function.parametric_domain);
  END_IF;
  IF 'IMPORTED_SURFACE_FUNCTION' IN typenames THEN
    RETURN (func\imported_surface_function.parametric_domain);
  END_IF;
  IF 'IMPORTED_VOLUME_FUNCTION' IN typenames THEN
    RETURN (func\imported_volume_function.parametric_domain);
  END_IF;
  IF 'APPLICATION_DEFINED_FUNCTION' IN typenames THEN
    RETURN (func\application_defined_function.explicit_domain);
  END_IF;
  -- Unreachable, as no other subtypes of maths_function are permissible without
  -- first modifying this function to account for them.
  RETURN (?);
END_FUNCTION;  -- derive_function_domain
(*
```

Argument definitions:

func: (input) The function whose domain is to be derived.

return: (output) An instance of **tuple_space** representing the domain of the function.

4.6.31 derive_function_range

This function returns an instance of **tuple_space** representing the range of the **maths_function** to which it is applied.

EXPRESS specification:

```
*)
FUNCTION derive_function_range(func : maths_function) : tuple_space;
  LOCAL
    typenames : SET OF STRING := stripped_typeof(func);
    tspace : tuple_space := make_listed_product_space ([]);
    m, n : nonnegative_integer := 0;
  END_LOCAL;
  IF 'FINITE_FUNCTION' IN typenames THEN
    RETURN (derive_finite_function_range (func\finite_function.pairs));
  END_IF;
  IF 'CONSTANT_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (make_finite_space ([func\constant_function.sole_output])));
  END_IF;
  IF 'SELECTOR_FUNCTION' IN typenames THEN
    tspace := func.domain;
    IF (space_dimension(tspace) = 1) AND ((schema_prefix + 'TUPLE_SPACE') IN
      TYPEOF (tspace)) THEN
      tspace := factor1 (tspace);
    END_IF;
    RETURN (one_tuples_of (factor_space (tspace, func\selector_function.selector)));
  END_IF;
  IF 'ELEMENTARY_FUNCTION' IN typenames THEN
    RETURN (derive_elementary_function_range (func\elementary_function.func_id));
```

© ISO 2002 — All rights reserved

```
  END_IF;
  IF 'RESTRICTION_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (func\restriction_function.operand));
  END_IF;
  IF 'REPACKAGING_FUNCTION' IN typenames THEN
    tspace := func\repackaging_function.operand.range;
    IF func\repackaging_function.output_repack = ro_wrap_as_tuple THEN
      tspace := one_tuples_of (tspace);
    END_IF;
    IF func\repackaging_function.output_repack = ro_unwrap_tuple THEN
      tspace := factor1 (tspace);
    END_IF;
    IF func\repackaging_function.selected_output > 0 THEN
      tspace := one_tuples_of (factor_space (tspace,
        func\repackaging_function.selected_output));
    END_IF;
    RETURN (tspace);
  END_IF;
  IF 'REINDEXED_ARRAY_FUNCTION' IN typenames THEN
    RETURN (func\unary_generic_expression.operand\maths_function.range);
  END_IF;
  IF 'SERIES_COMPOSED_FUNCTION' IN typenames THEN
    RETURN (func\series_composed_function.operands[SIZEOF
      (func\series_composed_function.operands)].range);
  END_IF;
  IF 'PARALLEL_COMPOSED_FUNCTION' IN typenames THEN
    RETURN (func\parallel_composed_function.final_function.range);
  END_IF;
  IF 'EXPLICIT_TABLE_FUNCTION' IN typenames THEN
    IF 'LISTED_REAL_DATA' IN typenames THEN
      RETURN (one_tuples_of (the_reals));
    END_IF;
    IF 'LISTED_INTEGER_DATA' IN typenames THEN
      RETURN (one_tuples_of (the_integers));
    END_IF;
    IF 'LISTED_LOGICAL_DATA' IN typenames THEN
      RETURN (one_tuples_of (the_logicals));
    END_IF;
    IF 'LISTED_STRING_DATA' IN typenames THEN
      RETURN (one_tuples_of (the_strings));
    END_IF;
    IF 'LISTED_COMPLEX_NUMBER_DATA' IN typenames THEN
      RETURN (one_tuples_of (the_complex_numbers));
    END_IF;
    IF 'LISTED_DATA' IN typenames THEN
      RETURN (one_tuples_of (func\listed_data.value_range));
    END_IF;
    IF 'EXTERNALLY_LISTED_DATA' IN typenames THEN
      RETURN (one_tuples_of (func\externally_listed_data.value_range));
    END_IF;
    IF 'LINEARIZED_TABLE_FUNCTION' IN typenames THEN
      RETURN (func\linearized_table_function.source.range);
    END_IF;
    IF 'BASIC_SPARSE_MATRIX' IN typenames THEN
      RETURN (func\basic_sparse_matrix.val.range);
    END_IF;
    -- Unreachable, as no other subtypes of explicit_table_function are permissible
    -- without first modifying this function to account for them.
    RETURN (?);
```

 © ISO 2002 — All rights reserved

```
  END_IF;
  IF 'HOMOGENEOUS_LINEAR_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (make_uniform_product_space
      (factor1 (func\homogeneous_linear_function.mat.range),
      func\homogeneous_linear_function.mat\explicit_table_function.shape
      [3 - func\homogeneous_linear_function.sum_index])));
  END_IF;
  IF 'GENERAL_LINEAR_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (make_uniform_product_space
      (factor1 (func\general_linear_function.mat.range),
      func\general_linear_function.mat\explicit_table_function.shape
      [3 - func\general_linear_function.sum_index])));
  END_IF;
  IF 'B_SPLINE_BASIS' IN typenames THEN
    RETURN (one_tuples_of (make_uniform_product_space (the_reals,
      func\b_spline_basis.num_basis)));
  END_IF;
  IF 'B_SPLINE_FUNCTION' IN typenames THEN
    tspace := factor1 (func\b_spline_function.coef.domain);
    m := SIZEOF (func\b_spline_function.basis);
    n := space_dimension (tspace);
    IF m = n THEN
      RETURN (one_tuples_of (the_reals));
    END_IF;
    IF m = n - 1 THEN
      RETURN (one_tuples_of (make_uniform_product_space (the_reals,
        factor_space (tspace, n)\finite_integer_interval.size)));
    END_IF;
    tspace := extract_factors (tspace, m+1, n);
    RETURN (one_tuples_of (make_function_space (sc_equal, tspace, sc_subspace,
      number_superspace_of (func\b_spline_function.coef.range))));
  END_IF;
  IF 'RATIONALIZE_FUNCTION' IN typenames THEN
    tspace := factor1 (func\rationalize_function.fun.range);
    n := space_dimension (tspace);
    RETURN (one_tuples_of (make_uniform_product_space (number_superspace_of (
      factor1 (tspace)), n-1)));
  END_IF;
  IF 'PARTIAL_DERIVATIVE_FUNCTION' IN typenames THEN
    RETURN (drop_numeric_constraints (
      func\partial_derivative_function.derivand.range));
  END_IF;
  IF 'DEFINITE_INTEGRAL_FUNCTION' IN typenames THEN
    RETURN (drop_numeric_constraints (
      func\definite_integral_function.integrand.range));
  END_IF;
  IF 'ABSTRACTED_EXPRESSION_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of(values_space_of(func\abstracted_expression_function.expr)));
  END_IF;
  IF 'EXPRESSION_DENOTED_FUNCTION' IN typenames THEN
    RETURN (values_space_of (func\expression_denoted_function.expr)\function_space.
      range_argument);
  END_IF;
  IF 'IMPORTED_POINT_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (make_uniform_product_space (the_reals,
      dimension_of (func\imported_point_function.geometry))));
  END_IF;
  IF 'IMPORTED_CURVE_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (make_uniform_product_space (the_reals,
```

© ISO 2002 — All rights reserved

```
      dimension_of (func\imported_curve_function.geometry))));
  END_IF;
  IF 'IMPORTED_SURFACE_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (make_uniform_product_space (the_reals,
      dimension_of (func\imported_surface_function.geometry))));
  END_IF;
  IF 'IMPORTED_VOLUME_FUNCTION' IN typenames THEN
    RETURN (one_tuples_of (make_uniform_product_space (the_reals,
      dimension_of (func\imported_volume_function.geometry))));
  END_IF;
  IF 'APPLICATION_DEFINED_FUNCTION' IN typenames THEN
    RETURN (func\application_defined_function.explicit_range);
  END_IF;
  -- Unreachable, as no other subtypes of maths_function are permissible without
  -- first modifying this function to account for them.
  RETURN (?);
END_FUNCTION;  -- derive_function_range
(*
```

Argument definitions:

func: (input) The function whose range is to be derived.

return: (output) An instance of **tuple_space** representing the range of the function.

4.6.32 domain_from

This function returns the domain from the **source_of_domain** attribute used by **constant_-function**, **selector_function** and **parallel_composed_function** instances.

EXPRESS specification:

```
*)
FUNCTION domain_from(ref : maths_space_or_function) : tuple_space;
  LOCAL
    typenames : SET OF STRING := stripped_typeof(ref);
    func      : maths_function;
  END_LOCAL;
  IF NOT EXISTS (ref) THEN  RETURN (?);  END_IF;
  IF 'TUPLE_SPACE' IN typenames THEN  RETURN (ref);                  END_IF;
  IF 'MATHS_SPACE' IN typenames THEN  RETURN (one_tuples_of (ref));  END_IF;
  func := ref;
  IF 'CONSTANT_FUNCTION' IN typenames THEN
    RETURN (domain_from (func\constant_function.source_of_domain));
  END_IF;
  IF 'SELECTOR_FUNCTION' IN typenames THEN
    RETURN (domain_from (func\selector_function.source_of_domain));
  END_IF;
  IF 'PARALLEL_COMPOSED_FUNCTION' IN typenames THEN
    RETURN (domain_from (func\parallel_composed_function.source_of_domain));
  END_IF;
  RETURN (func.domain);
END_FUNCTION;  -- domain_from
(*
```

 © ISO 2002 — All rights reserved

Argument definitions:

ref: (input) The source of the domain.

return: (output) The domain indicated by the source.

4.6.33 dot_count

This function scans its input string and returns the number of '.' characters which it contains.

EXPRESS specification:

```
*)
FUNCTION dot_count(str : STRING) : INTEGER;
  LOCAL
    n : INTEGER := 0;
  END_LOCAL;
  REPEAT i := 1 TO LENGTH (str);
    IF str[i] = '.' THEN  n := n + 1;  END_IF;
  END_REPEAT;
  RETURN (n);
END_FUNCTION;  -- dot_count
(*
```

Argument definitions:

str: (input) The string whose '.' characters are to be counted.

return: (output) The number of dot characters in the input string.

4.6.34 dotted_identifiers_syntax

This function determines whether or not a string has the syntax of an EXPRESS identifier or qualified identifier. That is, whether or not it is syntactically a sequence of one or more EXPRESS identifiers separated by periods (also known as full stops).

EXPRESS specification:

```
*)
FUNCTION dotted_identifiers_syntax(str : STRING) : BOOLEAN;
  LOCAL
    k : positive_integer;
    m : positive_integer;
  END_LOCAL;
  IF NOT EXISTS (str) THEN  RETURN (FALSE);  END_IF;
  k := parse_express_identifier (str, 1);
  IF k = 1 THEN  RETURN (FALSE);  END_IF;
  REPEAT WHILE k <= LENGTH (str);
    IF (str[k] <> '.') OR (k = LENGTH (str)) THEN  RETURN (FALSE);  END_IF;
    m := parse_express_identifier (str, k+1);
    IF m = k + 1 THEN  RETURN (FALSE);  END_IF;
    k := m;
```

© ISO 2002 — All rights reserved

```
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- dotted_identifiers_syntax
(*
```

Argument definitions:

str: (input) The string to be tested.

return: (output) A BOOLEAN value which is TRUE if the input string value is syntactically a sequence of identifiers separated by periods.

4.6.35 drop_numeric_constraints

This function returns a maths space corresponding in structure and component type to its input space, but with any numeric space component replaced by its corresponding unconstrained elementary space. This process is carried out recursively on the components of spaces which are tuple spaces.

NOTE This function is used to derive the function range for instances of **partial_derivative_function** and **definite_integral_function**.

EXPRESS specification:

```
*)
FUNCTION drop_numeric_constraints(spc : maths_space) : maths_space;
  LOCAL
    typenames : SET OF STRING := stripped_typeof(spc);
    tspc : listed_product_space;
    factors : LIST OF maths_space := [];
    xspc : extended_tuple_space;
  END_LOCAL;
  IF 'UNIFORM_PRODUCT_SPACE' IN typenames THEN
    RETURN (make_uniform_product_space (drop_numeric_constraints (
      spc\uniform_product_space.base), spc\uniform_product_space.exponent));
  END_IF;
  IF 'LISTED_PRODUCT_SPACE' IN typenames THEN
    tspc := spc;
    REPEAT i := 1 TO SIZEOF (tspc.factors);
      INSERT (factors, drop_numeric_constraints (tspc.factors[i]), i-1);
    END_REPEAT;
    RETURN (make_listed_product_space (factors));
  END_IF;
  IF 'EXTENDED_TUPLE_SPACE' IN typenames THEN
    xspc := spc;
    RETURN (make_extended_tuple_space (drop_numeric_constraints (xspc.base),
      drop_numeric_constraints (xspc.extender)));
  END_IF;
  IF subspace_of_es (spc, es_numbers) THEN
    RETURN (number_superspace_of (spc));
  END_IF;
  RETURN (spc);
END_FUNCTION;  -- drop_numeric_constraints
(*
```

 © ISO 2002 — All rights reserved

Argument definitions:

spc: (input) The tuple space from which numeric constraints are to be dropped.

return: (output) The tuple space corresponding to **spc** with its numeric constraints removed.

4.6.36 enclose_cregion_in_pregion

This function constructs the smallest instance of **polar_complex_number_region** with given centre which contains a given instance of **cartesian_complex_number_region**, if such an instance exists. It returns the indeterminate value if no such instance exists.

NOTE The indeterminate value is returned if either input is indeterminate or both the given centre is inside the given region and also the given region has infinite extent.

EXPRESS specification:

```
*)
FUNCTION enclose_cregion_in_pregion(crgn   : cartesian_complex_number_region;
                                    centre : complex_number_literal)
                                   : polar_complex_number_region;
  -- Find equivalent direction in range -PI < a <= PI.
  FUNCTION angle(a : REAL) : REAL;
    REPEAT WHILE a > PI;    a := a - 2.0*PI;  END_REPEAT;
    REPEAT WHILE a <= -PI;  a := a + 2.0*PI;  END_REPEAT;
    RETURN (a);
  END_FUNCTION;
  -- Determine whether a real is strictly within a real interval
  FUNCTION strictly_in(z    : REAL;
                       zitv : real_interval) : LOGICAL;
    RETURN ((NOT min_exists(zitv) OR (z > real_min(zitv))) AND
      (NOT max_exists(zitv) OR (z < real_max(zitv))));
  END_FUNCTION;
  -- Include direction in minmax collection
  PROCEDURE angle_minmax(    ab, a           : REAL;
                             a_in            : BOOLEAN;
                         VAR amin, amax      : REAL;
                         VAR amin_in, amax_in : BOOLEAN);
    a := angle(a - ab);
    IF amin = a THEN  amin_in := amin_in OR a_in;  END_IF;
    IF amin > a THEN  amin := a;  amin_in := a_in;  END_IF;
    IF amax = a THEN  amax_in := amax_in OR a_in;  END_IF;
    IF amax < a THEN  amax := a;  amax_in := a_in;  END_IF;
  END_PROCEDURE;
  -- Include distance in max collection
  PROCEDURE range_max(    r       : REAL;
                          incl    : BOOLEAN;
                      VAR rmax    : REAL;
                      VAR rmax_in : BOOLEAN);
    IF rmax = r THEN  rmax_in := rmax_in OR incl;   END_IF;
    IF rmax < r THEN  rmax := r;  rmax_in := incl;  END_IF;
  END_PROCEDURE;
  -- Include distance in min collection
  PROCEDURE range_min(    r       : REAL;
                          incl    : BOOLEAN;
                      VAR rmin    : REAL;
```

© ISO 2002 — All rights reserved

```
                  VAR rmin_in : BOOLEAN);
    IF rmin = r THEN  rmin_in := rmin_in OR incl;  END_IF;
    IF (rmin < 0.0) OR (rmin > r) THEN  rmin := r;  rmin_in := incl;  END_IF;
  END_PROCEDURE;
  LOCAL
    xitv, yitv : real_interval;
    is_xmin, is_xmax, is_ymin, is_ymax : BOOLEAN;
    xmin, xmax, ymin, ymax, xc, yc : REAL := 0.0;
    xmin_in, xmax_in, ymin_in, ymax_in : BOOLEAN := FALSE;
    rmin, rmax : REAL := -1.0;
    amin : REAL := 4.0;
    amax : REAL := -4.0;
    rmax_exists, outside : BOOLEAN := TRUE;
    rmin_in, rmax_in, amin_in, amax_in : BOOLEAN := FALSE;
    ab, a, r : REAL := 0.0;
    incl : BOOLEAN;
    ritv : real_interval;
    aitv : finite_real_interval;
    minclo, maxclo : open_closed := open;
  END_LOCAL;
  IF NOT EXISTS (crgn) OR NOT EXISTS (centre) THEN  RETURN (?);  END_IF;
  -- Extract elementary input information
  xitv := crgn.real_constraint;
  yitv := crgn.imag_constraint;
  xc := centre.real_part;
  yc := centre.imag_part;
  is_xmin := min_exists(xitv);
  is_xmax := max_exists(xitv);
  is_ymin := min_exists(yitv);
  is_ymax := max_exists(yitv);
  IF is_xmin THEN  xmin := real_min(xitv);  xmin_in := min_included(xitv);  END_IF;
  IF is_xmax THEN  xmax := real_max(xitv);  xmax_in := max_included(xitv);  END_IF;
  IF is_ymin THEN  ymin := real_min(yitv);  ymin_in := min_included(yitv);  END_IF;
  IF is_ymax THEN  ymax := real_max(yitv);  ymax_in := max_included(yitv);  END_IF;
  rmax_exists := is_xmin AND is_xmax AND is_ymin AND is_ymax;
  -- Identify base direction with respect to which all relevant directions lie
  -- within +/- 0.5*PI, or that the centre lies properly inside crgn.
  IF       is_xmin AND (xc <= xmin)  THEN  ab := 0.0;
  ELSE IF  is_ymin AND (yc <= ymin)  THEN  ab := 0.5*PI;
  ELSE IF  is_ymax AND (yc >= ymax)  THEN  ab := -0.5*PI;
  ELSE IF  is_xmax AND (xc >= xmax)  THEN  ab := PI;
  ELSE                                     outside := FALSE;
  END_IF;  END_IF;  END_IF;  END_IF;
  IF NOT outside AND NOT rmax_exists THEN
    RETURN (?);  -- No enclosing polar region exists (requires whole plane)
  END_IF;
  -- Identify any closest point on a side but not a corner.
  IF       is_xmin AND (xc <= xmin) AND strictly_in(yc,yitv) THEN
    rmin := xmin - xc;  rmin_in := xmin_in;
  ELSE IF  is_ymin AND (yc <= ymin) AND strictly_in(xc,xitv) THEN
    rmin := ymin - yc;  rmin_in := ymin_in;
  ELSE IF  is_ymax AND (yc >= ymax) AND strictly_in(xc,xitv) THEN
    rmin := yc - ymax;  rmin_in := ymax_in;
  ELSE IF  is_xmax AND (xc >= xmax) AND strictly_in(yc,yitv) THEN
    rmin := xc - xmax;  rmin_in := xmax_in;
  END_IF;  END_IF;  END_IF;  END_IF;
  IF is_xmin THEN
    IF is_ymin THEN  -- Consider lower left corner
      r := SQRT((xmin-xc)**2 + (ymin-yc)**2);
```

 © ISO 2002 — All rights reserved

```
      incl := xmin_in AND ymin_in;
      IF rmax_exists THEN  range_max(r,incl,rmax,rmax_in);  END_IF;
      IF outside THEN
        IF r > 0.0 THEN
          range_min(r,incl,rmin,rmin_in);
          a := angle(atan2(ymin-yc,xmin-xc) - ab);
          IF xc = xmin THEN  incl := xmin_in;  END_IF;
          IF yc = ymin THEN  incl := ymin_in;  END_IF;
          angle_minmax(ab,a,incl,amin,amax,amin_in,amax_in);
        ELSE  -- Centre at lower left corner
          rmin := 0.0;                rmin_in := xmin_in AND ymin_in;
          amin := angle(0.0-ab);      amin_in := ymin_in;
          amax := angle(0.5*PI-ab);   amax_in := xmin_in;
        END_IF;
      END_IF;
    ELSE IF xc <= xmin THEN  -- Consider points near (xmin, -infinity)
      angle_minmax(ab,-0.5*PI,(xc=xmin) AND xmin_in,amin,amax,amin_in,amax_in);
    END_IF;  END_IF;
    IF NOT is_ymax AND (xc <= xmin) THEN  -- Consider points near (xmin, +infinity)
      angle_minmax(ab,0.5*PI,(xc=xmin) AND xmin_in,amin,amax,amin_in,amax_in);
    END_IF;
  END_IF;
  IF is_ymin THEN
    IF is_xmax THEN  -- Consider lower right corner
      r := SQRT((xmax-xc)**2 + (ymin-yc)**2);
      incl := xmax_in AND ymin_in;
      IF rmax_exists THEN  range_max(r,incl,rmax,rmax_in);  END_IF;
      IF outside THEN
        IF r > 0.0 THEN
          range_min(r,incl,rmin,rmin_in);
          a := angle(atan2(ymin-yc,xmax-xc) - ab);
          IF xc = xmax THEN  incl := xmax_in;  END_IF;
          IF yc = ymin THEN  incl := ymin_in;  END_IF;
          angle_minmax(ab,a,incl,amin,amax,amin_in,amax_in);
        ELSE  -- Centre at lower right corner
          rmin := 0.0;                rmin_in := xmax_in AND ymin_in;
          amin := angle(0.5*PI-ab);   amin_in := ymin_in;
          amax := angle(PI-ab);       amax_in := xmax_in;
        END_IF;
      END_IF;
    ELSE IF yc <= ymin THEN  -- Consider points near (+infinity, ymin)
      angle_minmax(ab,0.0,(yc=ymin) AND ymin_in,amin,amax,amin_in,amax_in);
    END_IF;  END_IF;
    IF NOT is_xmin AND (yc <= ymin) THEN  -- Consider points near (-infinity, ymin)
      angle_minmax(ab,PI,(yc=ymin) AND ymin_in,amin,amax,amin_in,amax_in);
    END_IF;
  END_IF;
  IF is_xmax THEN
    IF is_ymax THEN  -- Consider upper right corner
      r := SQRT((xmax-xc)**2 + (ymax-yc)**2);
      incl := xmax_in AND ymax_in;
      IF rmax_exists THEN  range_max(r,incl,rmax,rmax_in);  END_IF;
      IF outside THEN
        IF r > 0.0 THEN
          range_min(r,incl,rmin,rmin_in);
          a := angle(atan2(ymax-yc,xmax-xc) - ab);
          IF xc = xmax THEN  incl := xmax_in;  END_IF;
          IF yc = ymax THEN  incl := ymax_in;  END_IF;
          angle_minmax(ab,a,incl,amin,amax,amin_in,amax_in);
```

© ISO 2002 — All rights reserved

```
      ELSE  -- Centre at lower left corner
        rmin := 0.0;                rmin_in := xmax_in AND ymax_in;
        amin := angle(-PI-ab);      amin_in := ymax_in;
        amax := angle(-0.5*PI-ab);  amax_in := xmax_in;
      END_IF;
    END_IF;
  ELSE IF xc >= xmax THEN  -- Consider points near (xmax, +infinity)
    angle_minmax(ab,0.5*PI,(xc=xmax) AND xmax_in,amin,amax,amin_in,amax_in);
  END_IF;  END_IF;
  IF NOT is_ymin AND (xc >= xmax) THEN  -- Consider points near (xmax, -infinity)
    angle_minmax(ab,-0.5*PI,(xc=xmax) AND xmax_in,amin,amax,amin_in,amax_in);
  END_IF;
END_IF;
IF is_ymax THEN
  IF is_xmin THEN  -- Consider upper left corner
    r := SQRT((xmin-xc)**2 + (ymax-yc)**2);
    incl := xmin_in AND ymax_in;
    IF rmax_exists THEN  range_max(r,incl,rmax,rmax_in);  END_IF;
    IF outside THEN
      IF r > 0.0 THEN
        range_min(r,incl,rmin,rmin_in);
        a := angle(atan2(ymax-yc,xmin-xc) - ab);
        IF xc = xmin THEN  incl := xmin_in;  END_IF;
        IF yc = ymax THEN  incl := ymax_in;  END_IF;
        angle_minmax(ab,a,incl,amin,amax,amin_in,amax_in);
      ELSE  -- Centre at lower right corner
        rmin := 0.0;                rmin_in := xmin_in AND ymax_in;
        amin := angle(0.5*PI-ab);   amin_in := ymax_in;
        amax := angle(PI-ab);       amax_in := xmin_in;
      END_IF;
    END_IF;
  ELSE IF yc >= ymax THEN  -- Consider points near (-infinity, ymax)
    angle_minmax(ab,PI,(yc=ymax) AND ymax_in,amin,amax,amin_in,amax_in);
  END_IF;  END_IF;
  IF NOT is_xmax AND (yc >= ymax) THEN  -- Consider points near (+infinity, ymax)
    angle_minmax(ab,0.0,(yc=ymax) AND ymax_in,amin,amax,amin_in,amax_in);
  END_IF;
END_IF;
IF outside THEN  -- Change direction origin from ab back to zero
  amin := angle(amin+ab);
  IF amin = PI THEN  amin := -PI;  END_IF;
  amax := angle(amax+ab);
  IF amax <= amin THEN  amax := amax + 2.0*PI;  END_IF;
ELSE
  amin := -PI;  amin_in := FALSE;
  amax := PI;   amax_in := FALSE;
END_IF;
IF amin_in THEN  minclo := closed;  END_IF;
IF amax_in THEN  maxclo := closed;  END_IF;
aitv := make_finite_real_interval(amin,minclo,amax,maxclo);
minclo := open;
IF rmin_in THEN  minclo := closed;  END_IF;
IF rmax_exists THEN
  maxclo := open;
  IF rmax_in THEN  maxclo := closed;  END_IF;
  ritv := make_finite_real_interval(rmin,minclo,rmax,maxclo);
ELSE
  ritv := make_real_interval_from_min(rmin,minclo);
END_IF;
```

 © ISO 2002 — All rights reserved

```
  RETURN (make_polar_complex_number_region(centre,ritv,aitv));
END_FUNCTION;  -- enclose_cregion_in_pregion
(*
```

Argument definitions:

crgn: (input) The cartesian complex number region to be enclosed.

centre: (input) The centre to be used for the enclosing region.

return: (output) The smallest polar complex region with the given centre enclosing the input region, if any such region exists.

4.6.37 enclose_pregion_in_cregion

This function constructs the smallest instance of **cartesian_complex_number_region** which contains a given instance of **polar_complex_number_region**, if such an instance exists. It returns the indeterminate value if no such instance exists.

NOTE The indeterminate value is returned if the input is indeterminate or the input region extends arbitrarily far in all four cartesian directions.

EXPRESS specification:

```
*)
FUNCTION enclose_pregion_in_cregion(prgn : polar_complex_number_region)
                                   : cartesian_complex_number_region;
  PROCEDURE nearest_good_direction(acart    : REAL;
                                   aitv     : finite_real_interval;
                                   VAR a    : REAL;
                                   VAR a_in : BOOLEAN);
    a := acart;                    a_in := TRUE;
    IF      a < aitv.min THEN
      -- a+2.0*PI > aitv.min automatically!
      IF a+2.0*PI < aitv.max THEN                               RETURN;  END_IF;
      IF a+2.0*PI = aitv.max THEN  a_in := max_included(aitv);  RETURN;  END_IF;
    ELSE IF a = aitv.min THEN      a_in := min_included(aitv);  RETURN;
    ELSE IF a < aitv.max THEN                                   RETURN;
    ELSE IF a = aitv.max THEN      a_in := max_included(aitv);  RETURN;
    END_IF;  END_IF;  END_IF;  END_IF;
    IF COS(acart - aitv.max) >= COS(acart - aitv.min) THEN
      a := aitv.max;               a_in := max_included(aitv);
    ELSE
      a := aitv.min;               a_in := min_included(aitv);
    END_IF;
  END_PROCEDURE;
  LOCAL
    xc, yc, xmin, xmax, ymin, ymax : REAL := 0.0;
    ritv, xitv, yitv : real_interval;
    aitv : finite_real_interval;
    xmin_exists, xmax_exists, ymin_exists, ymax_exists : BOOLEAN;
    xmin_in, xmax_in, ymin_in, ymax_in : BOOLEAN := FALSE;
    a, r : REAL := 0.0;
    a_in : BOOLEAN := FALSE;
    min_clo, max_clo : open_closed := open;
```

© ISO 2002 — All rights reserved

```
END_LOCAL;
IF NOT EXISTS (prgn) THEN  RETURN (?);  END_IF;
-- Extract elementary input data
xc := prgn.centre.real_part;
yc := prgn.centre.imag_part;
ritv := prgn.distance_constraint;
aitv := prgn.direction_constraint;
-- Determine xmin data
nearest_good_direction(PI,aitv,a,a_in);
IF COS(a) >= 0.0 THEN
  xmin_exists := TRUE;
  xmin := xc + real_min(ritv)*COS(a);
  xmin_in := a_in AND (min_included(ritv) OR (COS(a) = 0.0));
ELSE
  IF max_exists(ritv) THEN
    xmin_exists := TRUE;
    xmin := xc + real_max(ritv)*COS(a);
    xmin_in := a_in AND max_included(ritv);
  ELSE
    xmin_exists := FALSE;
  END_IF;
END_IF;
-- Determine xmax data
nearest_good_direction(0.0,aitv,a,a_in);
IF COS(a) <= 0.0 THEN
  xmax_exists := TRUE;
  xmax := xc + real_min(ritv)*COS(a);
  xmax_in := a_in AND (min_included(ritv) OR (COS(a) = 0.0));
ELSE
  IF max_exists(ritv) THEN
    xmax_exists := TRUE;
    xmax := xc + real_max(ritv)*COS(a);
    xmax_in := a_in AND max_included(ritv);
  ELSE
    xmax_exists := FALSE;
  END_IF;
END_IF;
-- Determine ymin data
nearest_good_direction(-0.5*PI,aitv,a,a_in);
IF SIN(a) >= 0.0 THEN
  ymin_exists := TRUE;
  ymin := yc + real_min(ritv)*SIN(a);
  ymin_in := a_in AND (min_included(ritv) OR (SIN(a) = 0.0));
ELSE
  IF max_exists(ritv) THEN
    ymin_exists := TRUE;
    ymin := yc + real_max(ritv)*SIN(a);
    ymin_in := a_in AND max_included(ritv);
  ELSE
    ymin_exists := FALSE;
  END_IF;
END_IF;
-- Determine ymax data
nearest_good_direction(0.5*PI,aitv,a,a_in);
IF SIN(a) <= 0.0 THEN
  ymax_exists := TRUE;
  ymax := yc + real_min(ritv)*SIN(a);
  ymax_in := a_in AND (min_included(ritv) OR (SIN(a) = 0.0));
ELSE
```

 © ISO 2002 — All rights reserved

```
    IF max_exists(ritv) THEN
      ymax_exists := TRUE;
      ymax := yc + real_max(ritv)*SIN(a);
      ymax_in := a_in AND max_included(ritv);
    ELSE
      ymax_exists := FALSE;
    END_IF;
  END_IF;
  -- Construct result
  IF NOT (xmin_exists OR xmax_exists OR ymin_exists OR ymax_exists) THEN
    RETURN (?);  -- No finite boundaries exist
  END_IF;
  -- Construct real_constraint
  IF xmin_exists THEN
    IF xmin_in THEN  min_clo := closed;  ELSE  min_clo := open;  END_IF;
    IF xmax_exists THEN
      IF xmax_in THEN  max_clo := closed;  ELSE  max_clo := open;  END_IF;
      xitv := make_finite_real_interval(xmin,min_clo,xmax,max_clo);
    ELSE
      xitv := make_real_interval_from_min(xmin,min_clo);
    END_IF;
  ELSE
    IF xmax_exists THEN
      IF xmax_in THEN  max_clo := closed;  ELSE  max_clo := open;  END_IF;
      xitv := make_real_interval_to_max(xmax,max_clo);
    ELSE
      xitv := the_reals;
    END_IF;
  END_IF;
  -- Construct imag_constraint
  IF ymin_exists THEN
    IF ymin_in THEN  min_clo := closed;  ELSE  min_clo := open;  END_IF;
    IF ymax_exists THEN
      IF ymax_in THEN  max_clo := closed;  ELSE  max_clo := open;  END_IF;
      yitv := make_finite_real_interval(ymin,min_clo,ymax,max_clo);
    ELSE
      yitv := make_real_interval_from_min(ymin,min_clo);
    END_IF;
  ELSE
    IF ymax_exists THEN
      IF ymax_in THEN  max_clo := closed;  ELSE  max_clo := open;  END_IF;
      yitv := make_real_interval_to_max(ymax,max_clo);
    ELSE
      yitv := the_reals;
    END_IF;
  END_IF;
  -- Construct cartesian region
  RETURN (make_cartesian_complex_number_region(xitv,yitv));
END_FUNCTION;  -- enclose_pregion_in_cregion
(*
```

Argument definitions:

prgn: (input) The polar complex number region to be enclosed.

return: (output) The smallest cartesian complex region enclosing the input region, if any such region exists.

© ISO 2002 — All rights reserved

4.6.38 enclose_pregion_in_pregion

This function constructs the smallest instance of **polar_complex_number_region** with given centre which contains a given instance of **polar_complex_number_region**, if such an instance exists. It returns the indeterminate value if no such instance exists.

NOTE The indeterminate value is returned if either input is indeterminate or both the given centre is inside the given region and also the given region has infinite extent.

EXPRESS specification:

```
*)
FUNCTION enclose_pregion_in_pregion(prgn   : polar_complex_number_region;
                                    centre : complex_number_literal)
                                   : polar_complex_number_region;
  -- Find equivalent direction in range -PI < a <= PI.
  FUNCTION angle(a : REAL) : REAL;
    REPEAT WHILE a > PI;    a := a - 2.0*PI;  END_REPEAT;
    REPEAT WHILE a <= -PI;  a := a + 2.0*PI;  END_REPEAT;
    RETURN (a);
  END_FUNCTION;
  -- Find proper limits for direction interval
  PROCEDURE angle_range(VAR amin, amax : REAL);
    amin := angle(amin);
    IF amin = PI THEN  amin := -PI;  END_IF;
    amax := angle(amax);
    IF amax <= amin THEN  amax := amax + 2.0*PI;  END_IF;
  END_PROCEDURE;
  -- Determine whether a direction is strictly within a direction interval
  FUNCTION strictly_in(a    : REAL;
                       aitv : finite_real_interval) : LOGICAL;
    a := angle(a);
    RETURN ({aitv.min < a < aitv.max} OR {aitv.min < a+2.0*PI < aitv.max});
  END_FUNCTION;
  -- Find min and max and related inclusion booleans among four candidates,
  -- using a base direction chosen to ensure the algebraic comparisons are valid.
  PROCEDURE find_aminmax(    ab,a0,a1,a2,a3  : REAL;
                             in0,in1,in2,in3 : BOOLEAN;
                         VAR amin,amax       : REAL;
                         VAR amin_in,amax_in : BOOLEAN);
    LOCAL
      a : REAL;
    END_LOCAL;
    amin := angle(a0-ab);                 amin_in := in0;
    amax := amin;                         amax_in := in0;
    a := angle(a1-ab);
    IF a = amin THEN                      amin_in := amin_in OR in1;  END_IF;
    IF a < amin THEN  amin := a;          amin_in := in1;             END_IF;
    IF a = amax THEN                      amax_in := amax_in OR in1;  END_IF;
    IF a > amax THEN  amax := a;          amax_in := in1;             END_IF;
    a := angle(a2-ab);
    IF a = amin THEN                      amin_in := amin_in OR in2;  END_IF;
    IF a < amin THEN  amin := a;          amin_in := in2;             END_IF;
    IF a = amax THEN                      amax_in := amax_in OR in2;  END_IF;
    IF a > amax THEN  amax := a;          amax_in := in2;             END_IF;
    a := angle(a3-ab);
    IF a = amin THEN                      amin_in := amin_in OR in3;  END_IF;
```

 © ISO 2002 — All rights reserved

```
    IF a < amin THEN  amin := a;           amin_in := in3;             END_IF;
    IF a = amax THEN                       amax_in := amax_in OR in3;  END_IF;
    IF a > amax THEN  amax := a;           amax_in := in3;             END_IF;
    amin := amin+ab;
    amax := amax+ab;
    angle_range(amin,amax);
  END_PROCEDURE;

  LOCAL
    ritp, ritv : real_interval;
    aitp, aitv : finite_real_interval;
    xp, yp, xc, yc, rmax, rmin, amin, amax, rc, acp, apc : REAL := 0.0;
    rmax_in, rmin_in, amin_in, amax_in : BOOLEAN := FALSE;
    rmxp, rmnp, x, y, r, a, ab, r0, a0, r1, a1, r2, a2, r3, a3 : REAL := 0.0;
    in0, in1, in2, in3, inn : BOOLEAN := FALSE;
    minclo, maxclo : open_closed := open;
  END_LOCAL;
  -- Extract elementary input information
  IF NOT EXISTS (prgn) OR NOT EXISTS (centre) THEN  RETURN (?);  END_IF;
  xp := prgn.centre.real_part;
  yp := prgn.centre.imag_part;
  ritp := prgn.distance_constraint;
  aitp := prgn.direction_constraint;
  xc := centre.real_part;
  yc := centre.imag_part;
  IF (xc = xp) AND (yc = yp) THEN  RETURN (prgn);  END_IF;
  rc := SQRT((xp-xc)**2 + (yp-yc)**2);
  acp := atan2(yp-yc,xp-xc);
  apc := atan2(yc-yp,xc-xp);
  rmnp := real_min(ritp);
  -- Analyse cases by existence of max distance and direction limits
  IF max_exists(ritp) THEN
    rmxp := real_max(ritp);
    IF aitp.max - aitp.min = 2.0*PI THEN
      -- annulus or disk, with or without slot or puncture
      inn := NOT max_included(aitp);  -- slot exists;
      a := angle(aitp.min);  -- slot direction
      rmax := rc+rmxp;                  rmax_in := max_included(ritp);
      IF inn AND (acp = a) THEN  rmax_in := FALSE;  END_IF;
      IF rc > rmxp THEN
        a0 := ASIN(rmxp/rc);
        amin := angle(acp-a0);          amin_in := max_included(ritp);
        IF amin = PI THEN  amin := -PI;  END_IF;
        amax := angle(acp+a0);          amax_in := amin_in;
        IF amax < amin THEN  amax := amax + 2.0*PI;  END_IF;
        rmin := rc-rmxp;                rmin_in := amin_in;
        IF inn THEN
          -- slotted case
          IF apc = a THEN  rmin_in := FALSE;  END_IF;
          IF angle(amin+0.5*PI) = a THEN  amin_in := FALSE;  END_IF;
          IF angle(amax-0.5*PI) = a THEN  amax_in := FALSE;  END_IF;
        END_IF;
      ELSE IF rc = rmxp THEN
        amin := angle(acp-0.5*PI);      amin_in := FALSE;
        IF amin = PI THEN  amin := -PI;  END_IF;
        amax := angle(acp+0.5*PI);      amax_in := FALSE;
        IF amax < amin THEN  amax := amax + 2.0*PI;  END_IF;
        rmin := 0.0;                    rmin_in := max_included(ritp);
        IF inn AND (apc = a) THEN  rmin_in := FALSE;  END_IF;
```

© ISO 2002 — All rights reserved

```
    ELSE IF rc > rmnp THEN
      IF inn AND (apc = a) THEN  -- in the slot
        rmin := 0.0;                    rmin_in := FALSE;
        amin := aitp.min;               amin_in := FALSE;
        amax := aitp.max;               amax_in := FALSE;
      ELSE
        rmin := 0.0;                    rmin_in := TRUE;
        amin := -PI;                    amin_in := FALSE;
        amax := PI;                     amax_in := TRUE;
      END_IF;
    ELSE
      rmin := rmnp-rc;                  rmin_in := min_included(ritp);
      amin := -PI;                      amin_in := FALSE;
      amax := PI;                       amax_in := TRUE;
      IF inn THEN  -- Special cases when aligned with slot
        IF apc = a THEN
                                        rmin_in := FALSE;
          amin := aitp.min;             amin_in := FALSE;
          amax := aitp.max;             amax_in := FALSE;
        ELSE IF acp = a THEN
          amin := aitp.min;             amin_in := FALSE;
          amax := aitp.max;             amax_in := FALSE;
        END_IF;  END_IF;
      END_IF;
    END_IF;  END_IF;  END_IF;
  ELSE  -- direction range < 2*PI
    -- Compute data for corners with respect to xc,yc
    x := xp + rmxp*cos(aitp.min) - xc;
    y := yp + rmxp*sin(aitp.min) - yc;
    r0 := SQRT(x**2 + y**2);
    in0 := max_included(ritp) AND min_included(aitp);
    IF r0 <> 0.0 THEN  a0 := atan2(y,x);  END_IF;
    x := xp + rmxp*cos(aitp.max) - xc;
    y := yp + rmxp*sin(aitp.max) - yc;
    r1 := SQRT(x**2 + y**2);
    in1 := max_included(ritp) AND max_included(aitp);
    IF r1 <> 0.0 THEN  a1 := atan2(y,x);  END_IF;
    x := xp + rmnp*cos(aitp.max) - xc;
    y := yp + rmnp*sin(aitp.max) - yc;
    r2 := SQRT(x**2 + y**2);
    in2 := min_included(ritp) AND max_included(aitp);
    IF r2 <> 0.0 THEN  a2 := atan2(y,x);  ELSE  a2 := a1;  in2 := in1;  END_IF;
    IF r1 = 0.0 THEN  a1 := a2;  in1 := in2;  END_IF;
    x := xp + rmnp*cos(aitp.min) - xc;
    y := yp + rmnp*sin(aitp.min) - yc;
    r3 := SQRT(x**2 + y**2);
    in3 := min_included(ritp) AND min_included(aitp);
    IF r3 <> 0.0 THEN  a3 := atan2(y,x);  ELSE  a3 := a0;  in3 := in0;  END_IF;
    IF r0 = 0.0 THEN  a0 := a3;  in0 := in3;  END_IF;
    IF rmnp = 0.0 THEN  in2 := min_included(ritp);  in3 := in2;  END_IF;
    IF (apc = angle(aitp.min)) OR (acp = angle(aitp.min)) THEN
      in0 := min_included(aitp);
      in3 := in0;
    ELSE IF (apc = angle(aitp.max)) OR (acp = angle(aitp.max)) THEN
      in1 := max_included(aitp);
      in2 := in1;
    END_IF;  END_IF;
    -- Find rmax
    IF strictly_in(acp,aitp) THEN
```

 © ISO 2002 — All rights reserved

```
  rmax := rc+rmxp;                  rmax_in := max_included(ritp);
ELSE
  rmax := r0;                       rmax_in := in0;
  IF rmax = r1 THEN                 rmax_in := rmax_in OR in1;  END_IF;
  IF rmax < r1 THEN  rmax := r1;    rmax_in := in1;             END_IF;
  IF rmax = r2 THEN                 rmax_in := rmax_in OR in2;  END_IF;
  IF rmax < r2 THEN  rmax := r2;    rmax_in := in2;             END_IF;
  IF rmax = r3 THEN                 rmax_in := rmax_in OR in3;  END_IF;
  IF rmax < r3 THEN  rmax := r3;    rmax_in := in3;             END_IF;
END_IF;
-- Find rmin
IF strictly_in(apc,aitp) THEN
  IF rc >= rmxp THEN
    rmin := rc-rmxp;                rmin_in := max_included(ritp);
  ELSE IF rc <= rmnp THEN
    rmin := rmnp-rc;                rmin_in := min_included(ritp);
  ELSE
    rmin := 0.0;                    rmin_in := TRUE;
  END_IF;  END_IF;
ELSE
  rmin := r0;                       rmin_in := in0;
  a := apc-aitp.min;
  r := rc*COS(a);
  IF {rmnp < r < rmxp} THEN  -- use nearest point on line segment
    rmin := rc*SIN(ABS(a));         rmin_in := min_included(aitp);
  END_IF;
  a := apc-aitp.max;
  r := rc*COS(a);
  IF {rmnp < r < rmxp} THEN  -- try nearest point on line segment
    r := rc*SIN(ABS(a));            inn := max_included(aitp);
    IF r = rmin THEN                rmin_in := rmin_in OR inn;  END_IF;
    IF r < rmin THEN  rmin := r;    rmin_in := inn;             END_IF;
  END_IF;
  IF r1 = rmin THEN                 rmin_in := rmin_in OR in1;  END_IF;
  IF r1 < rmin THEN  rmin := r1;    rmin_in := in1;             END_IF;
  IF r2 = rmin THEN                 rmin_in := rmin_in OR in2;  END_IF;
  IF r2 < rmin THEN  rmin := r2;    rmin_in := in2;             END_IF;
  IF r3 = rmin THEN                 rmin_in := rmin_in OR in3;  END_IF;
  IF r3 < rmin THEN  rmin := r3;    rmin_in := in3;             END_IF;
END_IF;
-- Find amin and amax, initially with respect to base direction ab.
IF rc >= rmxp THEN  -- outside outer circle
  ab := acp;
  find_aminmax(ab,a0,a1,a2,a3,in0,in1,in2,in3,amin,amax,amin_in,amax_in);
  a := ACOS(rmxp/rc);
  IF strictly_in(apc-a,aitp) THEN
    amin := ab-ASIN(rmxp/rc);       amin_in := max_included(ritp);
  END_IF;
  IF strictly_in(apc+a,aitp) THEN
    amax := ab+ASIN(rmxp/rc);       amax_in := max_included(ritp);
  END_IF;
  angle_range(amin,amax);
ELSE IF rc > rmnp THEN
  ab := angle(0.5*(aitp.min+aitp.max));  -- reference direction
  find_aminmax(ab,a0,a1,a2,a3,in0,in1,in2,in3,amin,amax,amin_in,amax_in);
ELSE
  -- Using base direction midway in prgn, compute all directions using
  -- values which ensure a3 < a2 and a0 < a1 algebraically.
  ab := angle(0.5*(aitp.min+aitp.max));  -- reference direction
```

© ISO 2002 — All rights reserved

```
        a0 := angle(a0-ab);
        a1 := angle(a1-ab);
        a2 := angle(a2-ab);
        a3 := angle(a3-ab);
        IF a3 > a2 THEN  a2 := a2 + 2.0*PI;  END_IF;
        IF a0 > a1 THEN  a0 := a0 + 2.0*PI;  END_IF;
        IF a3 < a0 THEN  amin := a3;      amin_in := in3;
        ELSE             amin := a0;      amin_in := in0;  END_IF;
        IF a2 > a1 THEN  amax := a2;      amax_in := in2;
        ELSE             amax := a1;      amax_in := in1;  END_IF;
        IF (amax - amin > 2.0*PI) OR
          ((amax - amin = 2.0*PI) AND (amin_in OR amax_in)) THEN
          -- Cannot see out
          amin := -PI;                    amin_in := FALSE;
          amax := PI;                     amax_in := TRUE;
        ELSE
          amin := amin + ab;
          amax := amax + ab;
          angle_range(amin,amax);
        END_IF;
      END_IF;  END_IF;
    END_IF;
    IF rmin_in THEN  minclo := closed;  END_IF;
    IF rmax_in THEN  maxclo := closed;  END_IF;
    ritv := make_finite_real_interval(rmin,minclo,rmax,maxclo);
  ELSE  -- Not max_exists(ritp)
    IF (rc > rmnp) AND strictly_in(apc,aitp) THEN
      RETURN (?);  -- No pregion exists.  (Would require whole plane.)
    END_IF;
    IF aitp.max - aitp.min = 2.0*PI THEN
      -- complement of disk, with or without slot
      a := angle(aitp.min);  -- slot direction
      IF rc > rmnp THEN  -- already excluded if not aligned with slot
        IF max_included(aitp) THEN
          RETURN (?);  -- No pregion exists.  (Would require whole plane.)
        END_IF;
        rmin := 0.0;                      rmin_in := FALSE;
        amin := aitp.min;                 amin_in := FALSE;
        amax := aitp.max;                 amax_in := FALSE;
      ELSE
        rmin := rmnp-rc;                  rmin_in := min_included(ritp);
        amin := -PI;                      amin_in := FALSE;
        amax := PI;                       amax_in := TRUE;
        IF NOT max_included(aitp) THEN  -- Special cases when aligned with slot
          IF apc = a THEN
                                          rmin_in := FALSE;
            amin := aitp.min;             amin_in := FALSE;
            amax := aitp.max;             amax_in := FALSE;
          ELSE IF acp = a THEN
            amin := aitp.min;             amin_in := FALSE;
            amax := aitp.max;             amax_in := FALSE;
          END_IF;  END_IF;
        END_IF;
      END_IF;
    ELSE  -- direction range < 2*PI
      -- Compute data for corners with respect to xc,yc (two at infinity)
      a0 := angle(aitp.min);
      in0 := FALSE;
      a1 := angle(aitp.max);
```

 © ISO 2002 — All rights reserved

```
in1 := FALSE;
x := xp + rmnp*cos(aitp.max) - xc;
y := yp + rmnp*sin(aitp.max) - yc;
r2 := SQRT(x**2 + y**2);
in2 := min_included(ritp) AND max_included(aitp);
IF r2 <> 0.0 THEN  a2 := atan2(y,x);  ELSE  a2 := a1;  in2 := in1;  END_IF;
x := xp + rmnp*cos(aitp.min) - xc;
y := yp + rmnp*sin(aitp.min) - yc;
r3 := SQRT(x**2 + y**2);
in3 := min_included(ritp) AND min_included(aitp);
IF r3 <> 0.0 THEN  a3 := atan2(y,x);  ELSE  a3 := a0;  in3 := in0;  END_IF;
IF rmnp = 0.0 THEN  in2 := min_included(ritp);  in3 := in2;  END_IF;
IF (apc = angle(aitp.min)) OR (acp = angle(aitp.min)) THEN
  in0 := min_included(aitp);
  in3 := in0;
ELSE IF (apc = angle(aitp.max)) OR (acp = angle(aitp.max)) THEN
  in1 := max_included(aitp);
  in2 := in1;
END_IF;  END_IF;
-- Find rmin
IF strictly_in(apc,aitp) THEN
  rmin := rmnp-rc;                  rmin_in := min_included(ritp);
ELSE
  rmin := r2;                       rmin_in := in2;
  a := apc-aitp.min;
  r := rc*COS(a);
  IF rmnp < r THEN  -- use nearest point on aitp.min ray
    rmin := rc*SIN(ABS(a));         rmin_in := min_included(aitp);
  END_IF;
  a := apc-aitp.max;
  r := rc*COS(a);
  IF rmnp < r THEN  -- try nearest point on aitp.max ray
    r := rc*SIN(ABS(a));            inn := max_included(aitp);
    IF r = rmin THEN                rmin_in := rmin_in OR inn;  END_IF;
    IF r < rmin THEN  rmin := r;    rmin_in := inn;             END_IF;
  END_IF;
  IF r3 = rmin THEN                 rmin_in := rmin_in OR in3;  END_IF;
  IF r3 < rmin THEN  rmin := r3;    rmin_in := in3;             END_IF;
END_IF;
-- Find amin and amax
ab := angle(0.5*(aitp.min+aitp.max));  -- reference direction
IF rc > rmnp THEN
  find_aminmax(ab,a0,a1,a2,a3,in0,in1,in2,in3,amin,amax,amin_in,amax_in);
ELSE
  -- Using base direction midway in prgn, compute all directions using
  -- values which ensure a3 < a2 and a0 < a1 algebraically.
  a0 := angle(a0-ab);
  a1 := angle(a1-ab);
  a2 := angle(a2-ab);
  a3 := angle(a3-ab);
  IF a3 > a2 THEN  a2 := a2 + 2.0*PI;  END_IF;
  IF a0 > a1 THEN  a0 := a0 + 2.0*PI;  END_IF;
  IF a3 < a0 THEN  amin := a3;      amin_in := in3;
  ELSE             amin := a0;      amin_in := in0;  END_IF;
  IF a2 > a1 THEN  amax := a2;      amax_in := in2;
  ELSE             amax := a1;      amax_in := in1;  END_IF;
  IF (amax - amin > 2.0*PI) OR
    ((amax - amin = 2.0*PI) AND (amin_in OR amax_in)) THEN
    -- Cannot see out
```

© ISO 2002 — All rights reserved

```
          amin := -PI;                    amin_in := FALSE;
          amax := PI;                     amax_in := TRUE;
          IF (rmin = 0.0) AND rmin_in THEN
            RETURN (?);  -- No pregion exists.  (Would require whole plane.)
          END_IF;
        ELSE
          amin := amin + ab;
          amax := amax + ab;
          angle_range(amin,amax);
        END_IF;
      END_IF;
    END_IF;
    IF rmin_in THEN  minclo := closed;  END_IF;
    ritv := make_real_interval_from_min(rmin,minclo);
  END_IF;
  minclo := open;  maxclo := open;
  IF amin_in THEN  minclo := closed;  END_IF;
  IF amax_in THEN  maxclo := closed;  END_IF;
  aitv := make_finite_real_interval(amin,minclo,amax,maxclo);
  -- Construct polar region
  RETURN (make_polar_complex_number_region(centre,ritv,aitv));
END_FUNCTION;  -- enclose_pregion_in_pregion
(*
```

Argument definitions:

prgn: (input) The polar complex number region to be enclosed.

centre: (input) The centre to be used for the enclosing region.

return: (output) The smallest polar complex region with the given centre enclosing the input region, if any such region exists.

4.6.39 equal_cregion_pregion

This function tests whether or not an instance of **cartesian_complex_number_region** describes the same subspace of the complex numbers as an instance of **polar_complex_number_region**.

NOTE The only cases where two such instances can be the same is if they describe quadrants or half spaces whose boundaries parallel the axes.

EXPRESS specification:

```
*)
FUNCTION equal_cregion_pregion(crgn : cartesian_complex_number_region;
                               prgn : polar_complex_number_region) : LOGICAL;
  LOCAL
    arng, amin, xc, yc : REAL;
    aitv, xitv, yitv : real_interval;
    c_in : BOOLEAN;
  END_LOCAL;
  IF NOT EXISTS (crgn) OR NOT EXISTS (prgn) THEN  RETURN (FALSE);  END_IF;
  IF max_exists(prgn.distance_constraint) THEN  RETURN (FALSE);  END_IF;
  IF real_min(prgn.distance_constraint) <> 0.0 THEN  RETURN (FALSE);  END_IF;
```

 © ISO 2002 — All rights reserved

```
c_in := min_included(prgn.distance_constraint);
aitv := prgn.direction_constraint;
amin := aitv.min;
arng := aitv.max - amin;
xc := prgn.centre.real_part;
yc := prgn.centre.imag_part;
xitv := crgn.real_constraint;
yitv := crgn.imag_constraint;
IF arng = 0.5*PI THEN
  IF amin = 0.0 THEN  -- quadrant to upper right
    RETURN (NOT max_exists(xitv) AND NOT max_exists(yitv) AND min_exists(xitv)
      AND min_exists(yitv) AND (real_min(xitv) = xc) AND (real_min(yitv) = yc)
      AND ((c_in AND min_included(aitv) AND max_included(aitv)
            AND min_included(xitv) AND min_included(yitv))
        OR (NOT c_in AND NOT min_included(aitv) AND max_included(aitv)
            AND min_included(xitv) AND NOT min_included(yitv))
        OR (NOT c_in AND min_included(aitv) AND NOT max_included(aitv)
            AND NOT min_included(xitv) AND min_included(yitv))
        OR (NOT c_in AND NOT min_included(aitv) AND NOT max_included(aitv)
            AND NOT min_included(xitv) AND NOT min_included(yitv))));
  END_IF;
  IF amin = 0.5*PI THEN  -- quadrant to upper left
    RETURN (max_exists(xitv) AND NOT max_exists(yitv) AND NOT min_exists(xitv)
      AND min_exists(yitv) AND (real_max(xitv) = xc) AND (real_min(yitv) = yc)
      AND ((c_in AND min_included(aitv) AND max_included(aitv)
            AND max_included(xitv) AND min_included(yitv))
        OR (NOT c_in AND NOT min_included(aitv) AND max_included(aitv)
            AND max_included(xitv) AND NOT min_included(yitv))
        OR (NOT c_in AND min_included(aitv) AND NOT max_included(aitv)
            AND NOT max_included(xitv) AND min_included(yitv))
        OR (NOT c_in AND NOT min_included(aitv) AND NOT max_included(aitv)
            AND NOT max_included(xitv) AND NOT min_included(yitv))));
  END_IF;
  IF amin = -PI THEN  -- quadrant to lower left
    RETURN (max_exists(xitv) AND max_exists(yitv) AND NOT min_exists(xitv)
      AND NOT min_exists(yitv) AND (real_max(xitv) = xc) AND (real_max(yitv) = yc)
      AND ((c_in AND min_included(aitv) AND max_included(aitv)
            AND max_included(xitv) AND max_included(yitv))
        OR (NOT c_in AND NOT min_included(aitv) AND max_included(aitv)
            AND max_included(xitv) AND NOT max_included(yitv))
        OR (NOT c_in AND min_included(aitv) AND NOT max_included(aitv)
            AND NOT max_included(xitv) AND max_included(yitv))
        OR (NOT c_in AND NOT min_included(aitv) AND NOT max_included(aitv)
            AND NOT max_included(xitv) AND NOT max_included(yitv))));
  END_IF;
  IF amin = -0.5*PI THEN  -- quadrant to lower right
    RETURN (NOT max_exists(xitv) AND max_exists(yitv) AND min_exists(xitv)
      AND NOT min_exists(yitv) AND (real_min(xitv) = xc) AND (real_max(yitv) = yc)
      AND ((c_in AND min_included(aitv) AND max_included(aitv)
            AND min_included(xitv) AND max_included(yitv))
        OR (NOT c_in AND NOT min_included(aitv) AND max_included(aitv)
            AND min_included(xitv) AND NOT max_included(yitv))
        OR (NOT c_in AND min_included(aitv) AND NOT max_included(aitv)
            AND NOT min_included(xitv) AND max_included(yitv))
        OR (NOT c_in AND NOT min_included(aitv) AND NOT max_included(aitv)
            AND NOT min_included(xitv) AND NOT max_included(yitv))));
  END_IF;
END_IF;
IF arng = PI THEN
```

© ISO 2002 — All rights reserved

```
    IF amin = 0.0 THEN  -- upper half space
      RETURN (NOT max_exists(xitv) AND NOT max_exists(yitv) AND NOT min_exists(xitv)
        AND min_exists(yitv) AND (real_min(yitv) = yc)
        AND ((c_in AND min_included(aitv) AND max_included(aitv)
              AND min_included(yitv))
          OR (NOT c_in AND NOT min_included(aitv) AND NOT max_included(aitv)
              AND NOT min_included(yitv))));
    END_IF;
    IF amin = 0.5*PI THEN  -- left half space
      RETURN (max_exists(xitv) AND NOT max_exists(yitv) AND NOT min_exists(xitv)
        AND NOT min_exists(yitv) AND (real_max(xitv) = xc)
        AND ((c_in AND min_included(aitv) AND max_included(aitv)
              AND max_included(xitv))
          OR (NOT c_in AND NOT min_included(aitv) AND NOT max_included(aitv)
              AND NOT max_included(xitv))));
    END_IF;
    IF amin = -PI THEN  -- lower half space
      RETURN (NOT max_exists(xitv) AND max_exists(yitv) AND NOT min_exists(xitv)
        AND NOT min_exists(yitv) AND (real_max(yitv) = yc)
        AND ((c_in AND min_included(aitv) AND max_included(aitv)
              AND max_included(yitv))
          OR (NOT c_in AND NOT min_included(aitv) AND NOT max_included(aitv)
              AND NOT max_included(yitv))));
    END_IF;
    IF amin = -0.5*PI THEN  -- right half space
      RETURN (NOT max_exists(xitv) AND NOT max_exists(yitv) AND min_exists(xitv)
        AND NOT min_exists(yitv) AND (real_min(xitv) = xc)
        AND ((c_in AND min_included(aitv) AND max_included(aitv)
              AND min_included(xitv))
          OR (NOT c_in AND NOT min_included(aitv) AND NOT max_included(aitv)
              AND NOT min_included(xitv))));
    END_IF;
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- equal_cregion_pregion
(*
```

Argument definitions:

crgn: (input) A cartesian complex number region.

prgn: (input) A polar complex number region.

return: (output) **TRUE** if the complex number regions are the same, **FALSE** if the regions are known to be different, or **UNKNOWN** if it is not easily decidable from the information at hand.

4.6.40 equal_maths_functions

This function performs a value equality test on the mathematical objects represented by the two **maths_function** arguments. The value **UNKNOWN** is returned if the information available is insufficient to decide.

NOTE This function is most likely to be decisive if the inputs have already been simplified by function **simplify_maths_functions**.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION equal_maths_functions(fun1, fun2 : maths_function) : LOGICAL;
  LOCAL
    cum : LOGICAL;
  END_LOCAL;
  IF fun1 = fun2 THEN  RETURN (TRUE);   END_IF;
  cum := equal_maths_spaces(fun1.domain,fun2.domain);
  IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
  cum := cum AND equal_maths_spaces(fun1.range,fun2.range);
  IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
  -- A lot of further analysis is possible, but not required.
  RETURN (UNKNOWN);
END_FUNCTION;  -- equal_maths_functions
(*
```

Argument definitions:

fun1: (input) The first **maths_function**.

fun2: (input) The second **maths_function**.

return: (output) **TRUE** if the mathematical objects represented are the same, **FALSE** if the objects are known to be different, or **UNKNOWN** if it is not easily decidable from the information at hand.

4.6.41 equal_maths_spaces

This function performs a value equality test on the mathematical objects represented by the two **maths_space** arguments. The value **UNKNOWN** is returned if the information available is insufficient to decide.

NOTE This function is most likely to be decisive if the inputs have already been simplified by function **simplify_maths_spaces**.

EXPRESS specification:

```
*)
FUNCTION equal_maths_spaces(spc1, spc2 : maths_space) : LOGICAL;
  LOCAL
    spc1types : SET OF STRING := stripped_typeof(spc1);
    spc2types : SET OF STRING := stripped_typeof(spc2);
    set1, set2 : SET OF maths_value;
    cum : LOGICAL := TRUE;
    base : maths_space;
    expnt : INTEGER;
    factors : LIST OF maths_space;
    factors2 : LIST OF maths_space;
    fs1, fs2 : function_space;
    cum2 : LOGICAL;
  END_LOCAL;
  IF spc1 = spc2 THEN  RETURN (TRUE);  END_IF;
  -- Consider cases where it is not yet certain that spc1 <> spc2.
```

© ISO 2002 — All rights reserved

```
IF 'FINITE_SPACE' IN spc1types THEN
  set1 := spc1\finite_space.members;
  IF 'FINITE_SPACE' IN spc2types THEN
    -- Members may have different but equivalent representations and in
    -- different orders.  May also have disguised repeats in same set of members.
    set2 := spc2\finite_space.members;
    REPEAT i := 1 TO SIZEOF (set1);
      cum := cum AND member_of (set1[i], spc2);
      IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
    END_REPEAT;
    IF cum = TRUE THEN
      REPEAT i := 1 TO SIZEOF (set2);
        cum := cum AND member_of (set2[i], spc1);
        IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
      END_REPEAT;
    END_IF;
    RETURN (cum);
  END_IF;
  IF 'FINITE_INTEGER_INTERVAL' IN spc2types THEN
    set2 := [];
    REPEAT i := spc2\finite_integer_interval.min TO spc2\finite_integer_interval.max;
      set2 := set2 + [i];
    END_REPEAT;
    RETURN (equal_maths_spaces(spc1,make_finite_space(set2)));
  END_IF;
END_IF;
IF ('FINITE_INTEGER_INTERVAL' IN spc1types) AND ('FINITE_SPACE' IN spc2types) THEN
  set1 := [];
  REPEAT i := spc1\finite_integer_interval.min TO spc1\finite_integer_interval.max;
    set1 := set1 + [i];
  END_REPEAT;
  RETURN (equal_maths_spaces(make_finite_space(set1),spc2));
END_IF;
IF ('CARTESIAN_COMPLEX_NUMBER_REGION' IN spc1types) AND
  ('POLAR_COMPLEX_NUMBER_REGION' IN spc2types) THEN
  -- Quadrants and half spaces have two representations
  RETURN (equal_cregion_pregion(spc1,spc2));
END_IF;
IF ('POLAR_COMPLEX_NUMBER_REGION' IN spc1types) AND
  ('CARTESIAN_COMPLEX_NUMBER_REGION' IN spc2types) THEN
  -- Quadrants and half spaces have two representations
  RETURN (equal_cregion_pregion(spc2,spc1));
END_IF;
IF 'UNIFORM_PRODUCT_SPACE' IN spc1types THEN
  base := spc1\uniform_product_space.base;
  expnt := spc1\uniform_product_space.exponent;
  IF 'UNIFORM_PRODUCT_SPACE' IN spc2types THEN
    IF expnt <> spc2\uniform_product_space.exponent THEN  RETURN (FALSE);  END_IF;
    RETURN (equal_maths_spaces(base,spc2\uniform_product_space.base));
  END_IF;
  IF 'LISTED_PRODUCT_SPACE' IN spc2types THEN
    factors := spc2\listed_product_space.factors;
    IF expnt <> SIZEOF (factors) THEN  RETURN (FALSE);  END_IF;
    REPEAT i := 1 TO SIZEOF (factors);
      cum := cum AND equal_maths_spaces(base,factors[i]);
      IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
    END_REPEAT;
    RETURN (cum);
  END_IF;
```

 © ISO 2002 — All rights reserved

```
  END_IF;
  IF 'LISTED_PRODUCT_SPACE' IN spc1types THEN
    factors := spc1\listed_product_space.factors;
    IF 'UNIFORM_PRODUCT_SPACE' IN spc2types THEN
      IF spc2\uniform_product_space.exponent <> SIZEOF (factors) THEN
        RETURN (FALSE);
      END_IF;
      base := spc2\uniform_product_space.base;
      REPEAT i := 1 TO SIZEOF (factors);
        cum := cum AND equal_maths_spaces(base,factors[i]);
        IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
      END_REPEAT;
      RETURN (cum);
    END_IF;
    IF 'LISTED_PRODUCT_SPACE' IN spc2types THEN
      factors2 := spc2\listed_product_space.factors;
      IF SIZEOF (factors) <> SIZEOF (factors2) THEN  RETURN (FALSE);  END_IF;
      REPEAT i := 1 TO SIZEOF (factors);
        cum := cum AND equal_maths_spaces(factors[i],factors2[i]);
        IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
      END_REPEAT;
      RETURN (cum);
    END_IF;
  END_IF;
  IF ('EXTENDED_TUPLE_SPACE' IN spc1types) AND
    ('EXTENDED_TUPLE_SPACE' IN spc2types) THEN
    RETURN (equal_maths_spaces(spc1\extended_tuple_space.extender,
      spc2\extended_tuple_space.extender) AND equal_maths_spaces(
      spc1\extended_tuple_space.base, spc2\extended_tuple_space.base));
  END_IF;
  IF ('FUNCTION_SPACE' IN spc1types) AND
    ('FUNCTION_SPACE' IN spc2types) THEN
    fs1 := spc1;
    fs2 := spc2;
    IF fs1.domain_constraint <> fs2.domain_constraint THEN
      IF (fs1.domain_constraint = sc_equal) OR (fs2.domain_constraint = sc_equal) THEN
        RETURN (FALSE);
      END_IF;
      IF (fs1.domain_constraint <> sc_subspace) THEN
        fs1 := spc2;
        fs2 := spc1;
      END_IF;
      IF (fs1.domain_constraint <> sc_subspace) OR
        (fs2.domain_constraint <> sc_member) THEN
        -- Safety check.  Should be unreachable.
        RETURN (UNKNOWN);
      END_IF;
      IF any_space_satisfies(fs1.domain_constraint,fs1.domain_argument) <>
        any_space_satisfies(fs2.domain_constraint,fs2.domain_argument) THEN
        RETURN (FALSE);
      END_IF;
      IF NOT ('FINITE_SPACE' IN stripped_typeof(fs2.domain_argument)) THEN
        RETURN (FALSE);
      END_IF;
      IF SIZEOF (['FINITE_SPACE','FINITE_INTEGER_INTERVAL'] *
        stripped_typeof(fs1.domain_argument)) = 0 THEN
        RETURN (FALSE);
      END_IF;
      -- Remaining cases too complex.
```

© ISO 2002 — All rights reserved

```
      RETURN (UNKNOWN);
    END_IF;
    cum := equal_maths_spaces(fs1.domain_argument,fs2.domain_argument);
    IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
    IF fs1.range_constraint <> fs2.range_constraint THEN
      IF (fs1.range_constraint = sc_equal) OR (fs2.range_constraint = sc_equal) THEN
        RETURN (FALSE);
      END_IF;
      IF (fs1.range_constraint <> sc_subspace) THEN
        fs1 := spc2;
        fs2 := spc1;
      END_IF;
      IF (fs1.range_constraint <> sc_subspace) OR
        (fs2.range_constraint <> sc_member) THEN
        -- Safety check.  Should be unreachable.
        RETURN (UNKNOWN);
      END_IF;
      IF any_space_satisfies(fs1.range_constraint,fs1.range_argument) <>
        any_space_satisfies(fs2.range_constraint,fs2.range_argument) THEN
        RETURN (FALSE);
      END_IF;
      IF NOT ('FINITE_SPACE' IN stripped_typeof(fs2.range_argument)) THEN
        RETURN (FALSE);
      END_IF;
      IF SIZEOF (['FINITE_SPACE','FINITE_INTEGER_INTERVAL'] *
        stripped_typeof(fs1.range_argument)) = 0 THEN
        RETURN (FALSE);
      END_IF;
      -- Remaining cases too complex.
      RETURN (UNKNOWN);
    END_IF;
    cum := cum AND equal_maths_spaces(fs1.range_argument,fs2.range_argument);
    RETURN (cum);
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- equal_maths_spaces
(*
```

Argument definitions:

spc1: (input) The first **maths_space**.

spc2: (input) The second **maths_space**.

return: (output) **TRUE** if the mathematical objects represented are the same, **FALSE** if the objects are known to be different, or **UNKNOWN** if it is not easily decidable from the information at hand.

4.6.42 equal_maths_values

This function performs a value equality test on the mathematical objects represented by the two **maths_value** arguments. The value **UNKNOWN** is returned if the information available is insufficient to decide.

NOTE This function is most likely to be decisive if the inputs have already been simplified by function **simplify_maths_value**.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION equal_maths_values(val1, val2 : maths_value) : LOGICAL;
  FUNCTION mem_of_vs(val1, val2 : maths_value) : LOGICAL;
    IF NOT has_values_space(val2) THEN RETURN (UNKNOWN);  END_IF;
    IF NOT member_of(val1,values_space_of(val2)) THEN  RETURN (FALSE);  END_IF;
    RETURN (UNKNOWN);
  END_FUNCTION;  -- mem_of_vs
  LOCAL
    types1, types2 : SET OF STRING;
    list1, list2 : LIST OF maths_value;
    cum : LOGICAL := TRUE;
  END_LOCAL;
  IF NOT EXISTS (val1) OR NOT EXISTS (val2) THEN  RETURN (FALSE);  END_IF;
  IF val1 = val2 THEN  RETURN (TRUE);  END_IF;
  types1 := stripped_typeof (val1);
  types2 := stripped_typeof (val2);
  IF ('MATHS_ATOM' IN types1) OR ('COMPLEX_NUMBER_LITERAL' IN types1) THEN
    IF 'MATHS_ATOM'             IN types2 THEN  RETURN (FALSE);  END_IF;
    IF 'COMPLEX_NUMBER_LITERAL' IN types2 THEN  RETURN (FALSE);  END_IF;
    IF 'LIST'                   IN types2 THEN  RETURN (FALSE);  END_IF;
    IF 'MATHS_SPACE'            IN types2 THEN  RETURN (FALSE);  END_IF;
    IF 'MATHS_FUNCTION'         IN types2 THEN  RETURN (FALSE);  END_IF;
    IF 'GENERIC_EXPRESSION' IN types2 THEN  RETURN (mem_of_vs(val1,val2)); END_IF;
    RETURN (UNKNOWN);
  END_IF;
  IF ('MATHS_ATOM' IN types2) OR ('COMPLEX_NUMBER_LITERAL' IN types2) THEN
    RETURN (equal_maths_values(val2,val1));
  END_IF;
  IF 'LIST' IN types1 THEN
    IF 'LIST' IN types2 THEN
      list1 := val1;
      list2 := val2;
      IF SIZEOF (list1) <> SIZEOF (list2) THEN  RETURN (FALSE);  END_IF;
      REPEAT i := 1 TO SIZEOF (list1);
        cum := cum AND equal_maths_values (list1[i], list2[i]);
        IF cum = FALSE THEN  RETURN (FALSE);  END_IF;
      END_REPEAT;
      RETURN (cum);
    END_IF;
    IF 'MATHS_SPACE'            IN types2 THEN  RETURN (FALSE);  END_IF;
    IF 'MATHS_FUNCTION'         IN types2 THEN  RETURN (FALSE);  END_IF;
    IF 'GENERIC_EXPRESSION' IN types2 THEN  RETURN (mem_of_vs(val1,val2)); END_IF;
    RETURN (UNKNOWN);
  END_IF;
  IF 'LIST' IN types2 THEN  RETURN (equal_maths_values(val2,val1));  END_IF;
  IF 'MATHS_SPACE' IN types1 THEN
    IF 'MATHS_SPACE' IN types2 THEN
      RETURN (equal_maths_spaces(val1,val2));
    END_IF;
    IF 'MATHS_FUNCTION'         IN types2 THEN  RETURN (FALSE);  END_IF;
    IF 'GENERIC_EXPRESSION' IN types2 THEN  RETURN (mem_of_vs(val1,val2)); END_IF;
    RETURN (UNKNOWN);
  END_IF;
  IF 'MATHS_SPACE' IN types2 THEN  RETURN (equal_maths_values(val2,val1));  END_IF;
  IF 'MATHS_FUNCTION' IN types1 THEN
    IF 'MATHS_FUNCTION' IN types2 THEN
      RETURN (equal_maths_functions(val1,val2));
```

© ISO 2002 — All rights reserved

```
    END_IF;
    IF 'GENERIC_EXPRESSION' IN types2 THEN  RETURN (mem_of_vs(val1,val2)); END_IF;
    RETURN (UNKNOWN);
  END_IF;
  IF 'MATHS_FUNCTION' IN types2 THEN  RETURN (equal_maths_values(val2,val1));  END_IF;
  IF ('GENERIC_EXPRESSION' IN types1) AND ('GENERIC_EXPRESSION' IN types2) THEN
    IF NOT has_values_space(val1) OR NOT has_values_space(val2) THEN
      RETURN (UNKNOWN);
    END_IF;
    IF NOT compatible_spaces(values_space_of(val1),values_space_of(val2)) THEN
      RETURN (FALSE);
    END_IF;
  END_IF;
  RETURN (UNKNOWN);
END_FUNCTION;  -- equal_maths_values
(*
```

Argument definitions:

val1: (input) The first **maths_value**.

val2: (input) The second **maths_value**.

return: (output) **TRUE** if the mathematical objects represented are the same, **FALSE** if the objects are known to be different, or **UNKNOWN** if it is not easily decidable from the information at hand.

4.6.43 es_subspace_of_es

This function determines whether an elsementary space is a subspace of another elementary space, both identified by values of **elementary_space_enumerators**.

EXPRESS specification:

```
*)
FUNCTION es_subspace_of_es(es1, es2 : elementary_space_enumerators) : BOOLEAN;
  IF NOT EXISTS (es1) OR NOT EXISTS (es2) THEN  RETURN (FALSE);  END_IF;
  IF es1 = es2 THEN  RETURN (TRUE);  END_IF;
  IF es2 = es_generics THEN  RETURN (TRUE);  END_IF;
  IF (es1 = es_booleans) AND (es2 = es_logicals) THEN  RETURN (TRUE);  END_IF;
  IF (es2 = es_numbers) AND ((es1 = es_complex_numbers) OR (es1 = es_reals) OR
    (es1 = es_integers)) THEN  RETURN (TRUE);  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- es_subspace_of_es
(*
```

Argument definitions:

es1: (input) The enumeration value identifying the first elementary space.

es2: (input) The enumeration value identifying the second elementary space.

return: (output) A BOOLEAN value which is TRUE if the first space is a subspace of the second space, and FALSE, otherwise.

 © ISO 2002 — All rights reserved

4.6.44 expression_is_constant

This function determines whether an instance of **generic_expression** is a constant expression, that is, denotes a single object. A constant expression is identified by the fact that it contains no free variables.

EXPRESSION specification:

```
*)
FUNCTION expression_is_constant(expr : generic_expression) : BOOLEAN;
  RETURN (bool(SIZEOF (free_variables_of (expr)) = 0));
END_FUNCTION;  -- expression_is_constant
(*
```

Argument definitions:

expr: (input) The expression to be tested.

return: (output) The BOOLEAN indication of whether or not the expression is constant.

4.6.45 extract_factors

This function forms a tuple space from a sequence of consecutive factors of another tuple space.

EXPRESS specification:

```
*)
FUNCTION extract_factors(tspace : tuple_space;
                         m, n   : INTEGER) : tuple_space;
  LOCAL
    tsp : tuple_space := the_zero_tuple_space;
  END_LOCAL;
  REPEAT i := m TO n;
    tsp := assoc_product_space (tsp, factor_space (tspace, i));
  END_REPEAT;
  RETURN (tsp);
END_FUNCTION;  -- extract_factors
(*
```

Argument definitions:

tspace: (input) The tuple space from which factor spaces are to be extracted.

m: (input) The index of the first factor space to be extracted.

n: (input) The index of the last factor space to be extracted.

return: (output) The tuple space formed by taking the **m**th to **n**th factors from **tspace**.

© ISO 2002 — All rights reserved

4.6.46 extremal_position_check

This function verifies that the extreme locations for instances of **standard_table_function** and **regular_table_function** lie wihin the domain of the simple array function which actually supplies the values.

EXPRESS specification:

```
*)
FUNCTION extremal_position_check(fun : linearized_table_function) : BOOLEAN;
  LOCAL
    source_domain : maths_space;
    source_interval : finite_integer_interval;
    index : INTEGER := 1;
    base : INTEGER;
    shape : LIST OF positive_integer;
    ndim : positive_integer;
    slo, shi : INTEGER;
    sublo : LIST OF INTEGER := [];
    subhi : LIST OF INTEGER := [];
  END_LOCAL;
  IF NOT EXISTS (fun) THEN  RETURN (FALSE);  END_IF;
  source_domain := factor1 (fun.source.domain);
  IF (schema_prefix + 'TUPLE_SPACE') IN TYPEOF (source_domain) THEN
    source_domain := factor1 (source_domain);
  END_IF;
  IF NOT ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (source_domain)) THEN
    RETURN (FALSE);
  END_IF;
  source_interval := source_domain;
  base := fun\explicit_table_function.index_base;
  shape := fun\explicit_table_function.shape;
  IF (schema_prefix + 'STANDARD_TABLE_FUNCTION') IN TYPEOF (fun) THEN
    REPEAT j := 1 TO SIZEOF (shape);
      index := index * shape[j];
    END_REPEAT;
    index := fun.first + index - 1;
    RETURN (bool({source_interval.min <= index <= source_interval.max}));
  END_IF;
  IF (schema_prefix + 'REGULAR_TABLE_FUNCTION') IN TYPEOF (fun) THEN
    ndim := SIZEOF (fun\explicit_table_function.shape);
    REPEAT j:= 1 TO ndim;
      slo := base;
      shi := base + shape[j] - 1;
      IF fun\regular_table_function.increments[j] >= 0 THEN
        INSERT (sublo, slo, j-1);
        INSERT (subhi, shi, j-1);
      ELSE
        INSERT (sublo, shi, j-1);
        INSERT (subhi, slo, j-1);
      END_IF;
    END_REPEAT;
    index := regular_indexing (sublo, base, shape,
      fun\regular_table_function.increments, fun.first);
    IF NOT ({source_interval.min <= index <= source_interval.max}) THEN
      RETURN (FALSE);
    END_IF;
```

 © ISO 2002 — All rights reserved

```
    index := regular_indexing (subhi, base, shape,
      fun\regular_table_function.increments, fun.first);
    IF NOT ({source_interval.min <= index <= source_interval.max}) THEN
      RETURN (FALSE);
    END_IF;
    RETURN (TRUE);
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- extremal_position_check
(*
```

Argument definitions:

fun: (input) The instance of **linearized_table_function** to be checked.

return: (output) A BOOLEAN value which is TRUE if the extremal possible positions computed for a standard or regular table function lie within the domain of the associated values function.

4.6.47 factor1

This function returns the first factor space of a tuple space.

EXPRESS specification:

```
*)
FUNCTION factor1(tspace : tuple_space) : maths_space;
  LOCAL
    typenames : SET OF STRING := TYPEOF (tspace);
  END_LOCAL;
  IF (schema_prefix + 'UNIFORM_PRODUCT_SPACE') IN typenames THEN
    RETURN (tspace\uniform_product_space.base);
  END_IF;
  IF (schema_prefix + 'LISTED_PRODUCT_SPACE') IN typenames THEN
    RETURN (tspace\listed_product_space.factors[1]);
    -- This path could return the indeterminate value if the list is empty.
    -- This is the correct result for this case.
  END_IF;
  IF (schema_prefix + 'EXTENDED_TUPLE_SPACE') IN typenames THEN
    RETURN (factor1 (tspace\extended_tuple_space.base));
  END_IF;
  -- Should not be reachable.
  RETURN (?);
END_FUNCTION;  -- factor1
(*
```

Argument definitions:

tspace: (input) The input tuple space.

return: (output) The first factor space of the input tuple space.

© ISO 2002 — All rights reserved

4.6.48 factor_space

This function returns the selected factor space of a tuple space. If the selection index is out of range, the indeterminate value (?) is returned

EXPRESS specification:

```
*)
FUNCTION factor_space(tspace : tuple_space;
                      idx    : positive_integer) : maths_space;
  LOCAL
    typenames : SET OF STRING := TYPEOF (tspace);
  END_LOCAL;
  IF (schema_prefix + 'UNIFORM_PRODUCT_SPACE') IN typenames THEN
    IF idx <= tspace\uniform_product_space.exponent THEN
      RETURN (tspace\uniform_product_space.base);
    END_IF;
    RETURN (?);
  END_IF;
  IF (schema_prefix + 'LISTED_PRODUCT_SPACE') IN typenames THEN
    IF idx <= SIZEOF (tspace\listed_product_space.factors) THEN
      RETURN (tspace\listed_product_space.factors[idx]);
    END_IF;
    RETURN (?);
  END_IF;
  IF (schema_prefix + 'EXTENDED_TUPLE_SPACE') IN typenames THEN
    IF idx <= space_dimension (tspace\extended_tuple_space.base) THEN
      RETURN (factor_space (tspace\extended_tuple_space.base, idx));
    END_IF;
    RETURN (tspace\extended_tuple_space.extender);
  END_IF;
  -- Should not be reachable.
  RETURN (?);
END_FUNCTION;  -- factor_space
(*
```

Argument definitions:

tspace: (input) The input tuple space.

idx: (input) The index of the factor space to select.

return: (output) The selected factor space of the input tuple space.

4.6.49 free_variables_of

This function returns the set of free variables present in a **generic_expression** instance. It is similar to the **ISO13584_generic_expressions_schema.used_variables** function except that variables bound by instances of **quantifier_expression** are removed.

EXPRESS specification:

```
*)
```

 © ISO 2002 — All rights reserved

```
FUNCTION free_variables_of(expr : generic_expression) : SET OF generic_variable;
  LOCAL
    typenames : SET OF STRING := stripped_typeof(expr);
    result : SET OF generic_variable := [];
    exprs : LIST OF generic_expression := [];
  END_LOCAL;
  IF 'GENERIC_LITERAL' IN typenames THEN
    RETURN (result);
  END_IF;
  IF 'GENERIC_VARIABLE' IN typenames THEN
    result := result + expr;
    RETURN (result);
  END_IF;
  IF 'QUANTIFIER_EXPRESSION' IN typenames THEN
    exprs := QUERY (ge <* expr\multiple_arity_generic_expression.operands |
      NOT (ge IN expr\quantifier_expression.variables));
    REPEAT i := 1 TO SIZEOF (exprs);
      result := result + free_variables_of (exprs[i]);
    END_REPEAT;
    REPEAT i := 1 TO SIZEOF (expr\quantifier_expression.variables);
      result := result - expr\quantifier_expression.variables[i];
    END_REPEAT;
    RETURN (result);
  END_IF;
  IF 'UNARY_GENERIC_EXPRESSION' IN typenames THEN
    RETURN (free_variables_of (expr\unary_generic_expression.operand));
  END_IF;
  IF 'BINARY_GENERIC_EXPRESSION' IN typenames THEN
    result := free_variables_of (expr\binary_generic_expression.operands[1]);
    RETURN (result + free_variables_of (expr\binary_generic_expression.operands[2]));
  END_IF;
  IF 'MULTIPLE_ARITY_GENERIC_EXPRESSION' IN typenames THEN
    REPEAT i := 1 TO SIZEOF (expr\multiple_arity_generic_expression.operands);
      result := result + free_variables_of (
        expr\multiple_arity_generic_expression.operands[i]);
    END_REPEAT;
    RETURN (result);
  END_IF;
  -- In this case the subtype shall not contain any variable (see IP1 in
  -- generic_expression).
  RETURN (result);
END_FUNCTION;  -- free_variables_of
(*
```

Argument definitions:

expr: (input) The expression from which the free variables are to be extracted.

return: (output) The set of free variables in the expression.

4.6.50 function_applicability

This function verifies that the domain space of a function and the arguments to which the function are being applied are compatible. This means that the number of arguments is suitable and that the value space of each expression being supplied as an argument is not known to have empty intersection with the corresponding factor space of the function domain.

© ISO 2002 — All rights reserved

NOTE Observe that satisfying the function applicability test does not guarantee that the actual arguments are in the function domain. The purpose of this test is to rule out the obviously ridiculous cases.

EXPRESS specification:

```
*)
FUNCTION function_applicability(func      : maths_function_select;
                                arguments : LIST [1:?] OF maths_value) : BOOLEAN;
  LOCAL
    domain : tuple_space := convert_to_maths_function(func).domain;
    domain_types : SET OF STRING := TYPEOF (domain);
    narg : positive_integer := SIZEOF (arguments);
    arg : generic_expression;
  END_LOCAL;
  IF (schema_prefix + 'PRODUCT_SPACE') IN domain_types THEN
    IF space_dimension (domain) <> narg THEN  RETURN (FALSE);  END_IF;
  ELSE
    IF (schema_prefix + 'EXTENDED_TUPLE_SPACE') IN domain_types THEN
      IF space_dimension (domain) > narg THEN  RETURN (FALSE);  END_IF;
    ELSE
      RETURN (FALSE);  -- Should be unreachable
    END_IF;
  END_IF;
  REPEAT i := 1 TO narg;
    arg := convert_to_operand (arguments[i]);
    IF NOT has_values_space (arg) THEN  RETURN (FALSE);  END_IF;
    IF NOT compatible_spaces (factor_space (domain, i), values_space_of (arg)) THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- function_applicability
(*
```

Argument definitions:

func: (input) The function as a value of type **maths_function_select**.

arguments: (input) The arguments to the function.

return: (output) A BOOLEAN value of FALSE is returned if the domain is known to be incompatible with the argument list in number or component space. Otherwise, TRUE is returned.

4.6.51 function_is_1d_array

This function determines whether or not its argument is a one-dimensional array function. That is, whether the sole input to **func** is an integer from a finite integer interval or a one-tuple of integers from a finite integer interval.

NOTE The acceptance of these two options reflects an assumption of automatic repackaging as needed in the uses of the argument.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION function_is_1d_array(func : maths_function) : BOOLEAN;
  LOCAL
    temp : maths_space;
  END_LOCAL;
  IF NOT EXISTS (func) THEN  RETURN (FALSE);  END_IF;
  IF space_dimension (func.domain) <> 1 THEN  RETURN (FALSE);  END_IF;
  temp := factor1 (func.domain);
  IF (schema_prefix + 'PRODUCT_SPACE') IN TYPEOF (temp) THEN
    IF space_dimension (temp) <> 1 THEN  RETURN (FALSE);  END_IF;
    temp := factor1 (temp);
  END_IF;
  IF (schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (temp) THEN
    RETURN (TRUE);
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- function_is_1d_array
(*
```

Argument definitions:

func: (input) The **maths_function** instance to be tested.

return: (output) A BOOLEAN value which is TRUE if the input is a one-dimensional array function, and FALSE, otherwise.

4.6.52 function_is_1d_table

This function determines whether or not its argument is a one-dimensional table function. That is, whether the sole input to **func** is an integer from a finite integer interval starting at zero or one, or a one-tuple of integers from a finite integer interval starting at zero or one.

NOTE The acceptance of these two options reflects an assumption of automatic repackaging as needed in the uses of the argument.

EXPRESS specification:

```
*)
FUNCTION function_is_1d_table(func : maths_function) : BOOLEAN;
  LOCAL
    temp : maths_space;
    itvl : finite_integer_interval;
  END_LOCAL;
  IF NOT EXISTS (func) THEN  RETURN (FALSE);  END_IF;
  IF space_dimension (func.domain) <> 1 THEN  RETURN (FALSE);  END_IF;
  temp := factor1 (func.domain);
  IF (schema_prefix + 'PRODUCT_SPACE') IN TYPEOF (temp) THEN
    IF space_dimension (temp) <> 1 THEN  RETURN (FALSE);  END_IF;
    temp := factor1 (temp);
  END_IF;
  IF (schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (temp) THEN
    itvl := temp;
```

© ISO 2002 — All rights reserved

```
    RETURN (bool((itvl.min = 0) OR (itvl.min = 1)));
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- function_is_1d_table
(*
```

Argument definitions:

func: (input) The **maths_function** instance to be tested.

return: (output) A BOOLEAN value which is TRUE if the input is a one-dimensional table function, and FALSE, otherwise.

4.6.53 function_is_2d_table

This function determines whether or not its argument is a two-dimensional table function. That is, whether the sole input to **func** is a pair of integers from a two-dimensional subscript space.

EXPRESS specification:

```
*)
FUNCTION function_is_2d_table(func : maths_function) : BOOLEAN;
  LOCAL
    temp : maths_space;
    pspace : product_space;
    itvl1, itvl2 : finite_integer_interval;
  END_LOCAL;
  IF NOT EXISTS (func) THEN  RETURN (FALSE);  END_IF;
  IF space_dimension (func.domain) <> 1 THEN  RETURN (FALSE);  END_IF;
  temp := factor1 (func.domain);
  IF NOT ('PRODUCT_SPACE' IN stripped_typeof(temp)) THEN  RETURN (FALSE);  END_IF;
  pspace := temp;
  IF space_dimension (pspace) <> 2 THEN  RETURN (FALSE);  END_IF;
  temp := factor1 (pspace);
  IF NOT ('FINITE_INTEGER_INTERVAL' IN stripped_typeof(temp)) THEN
    RETURN (FALSE);
  END_IF;
  itvl1 := temp;
  temp := factor_space (pspace, 2);
  IF NOT ('FINITE_INTEGER_INTERVAL' IN stripped_typeof(temp)) THEN
    RETURN (FALSE);
  END_IF;
  itvl2 := temp;
  RETURN (bool((itvl1.min = itvl2.min) AND ((itvl1.min = 0) OR (itvl1.min = 1))));
END_FUNCTION;  -- function_is_2d_table
(*
```

Argument definitions:

func: (input) The **maths_function** instance to be tested.

return: (output) A BOOLEAN value which is TRUE if the input is a two-dimensional table function, and FALSE, otherwise.

 © ISO 2002 — All rights reserved

4.6.54 function_is_array

This function determines whether an instance of **maths_function** is an array function. In other words, it determines whether the input to the function is a fixed number of integers whose tuple belongs to a subscript space, or a single object which is a tuple of integers from a subscript space. A subscript space is a finite cartesian product of finite integer intervals all of which start at zero or all of which start at one.

NOTE Acceptance of both types of input packaging implies that any necessary repackaging is done automatically where these functions are used.

EXPRESS specification:

```
*)
FUNCTION function_is_array(func : maths_function) : BOOLEAN;
  LOCAL
    tspace : tuple_space;
    temp : maths_space;
  END_LOCAL;
  IF NOT EXISTS (func) THEN  RETURN (FALSE);  END_IF;
  tspace := func.domain;
  IF (space_dimension (tspace) = 1) AND ((schema_prefix + 'TUPLE_SPACE') IN
    TYPEOF (factor1 (tspace))) THEN
    tspace := factor1 (tspace);
  END_IF;
  IF NOT ((schema_prefix + 'PRODUCT_SPACE') IN TYPEOF (tspace)) THEN
    RETURN (FALSE);
  END_IF;
  REPEAT i := 1 TO space_dimension (tspace);
    temp := factor_space (tspace, i);
    IF NOT ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (temp)) THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- function_is_array
(*
```

Argument definitions:

func: (input) The instance of **maths_function** to be tested.

return: (output) A BOOLEAN value which is TRUE if the function is an array function.

4.6.55 function_is_table

This function determines whether an instance of **maths_function** is a table function. In other words, it determines whether the input to the function is a fixed number of integers whose tuple belongs to a subscript space, or a single object which is a tuple of integers from a subscript space. A subscript space is a finite cartesian product of finite integer intervals all of which start at zero or all of which start at one.

NOTE Acceptance of both types of input packaging implies that any necessary repackaging is done automatically where these functions are used.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION function_is_table(func : maths_function) : BOOLEAN;
  LOCAL
    tspace : tuple_space;
    temp : maths_space;
    base : INTEGER;
  END_LOCAL;
  IF NOT EXISTS (func) THEN  RETURN (FALSE);  END_IF;
  tspace := func.domain;
  IF (space_dimension (tspace) = 1) AND ((schema_prefix + 'TUPLE_SPACE') IN
    TYPEOF (factor1 (tspace))) THEN
    tspace := factor1 (tspace);
  END_IF;
  IF NOT ((schema_prefix + 'PRODUCT_SPACE') IN TYPEOF (tspace)) THEN
    RETURN (FALSE);
  END_IF;
  temp := factor1 (tspace);
  IF NOT ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (temp)) THEN
    RETURN (FALSE);
  END_IF;
  base := temp\finite_integer_interval.min;
  IF (base <> 0) AND (base <> 1) THEN
    RETURN (FALSE);
  END_IF;
  REPEAT i := 2 TO space_dimension (tspace);
    temp := factor_space (tspace, i);
    IF NOT ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (temp)) THEN
      RETURN (FALSE);
    END_IF;
    IF temp\finite_integer_interval.min <> base THEN  RETURN (FALSE);  END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- function_is_table
(*
```

Argument definitions:

func: (input) The instance of **maths_function** to be tested.

return: (output) A BOOLEAN value which is TRUE if the function is a table function.

4.6.56 has_values_space

This function determines whether a **generic_expression** instance is one which the **mathematical_functions_schema** can work with. In particular, it identifies the expressions for which the function **values_space_of** can produce an output.

EXPRESS specification:

```
*)
FUNCTION has_values_space(expr : generic_expression) : BOOLEAN;
  LOCAL
```

 © ISO 2002 — All rights reserved

```
    typenames : SET OF STRING := stripped_typeof (expr);
  END_LOCAL;
  IF 'EXPRESSION' IN typenames THEN
    RETURN (bool(('NUMERIC_EXPRESSION' IN typenames) OR
      ('STRING_EXPRESSION' IN typenames) OR
      ('BOOLEAN_EXPRESSION' IN typenames)));
  END_IF;
  IF 'MATHS_FUNCTION' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'FUNCTION_APPLICATION' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'MATHS_SPACE' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'MATHS_VARIABLE' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'DEPENDENT_VARIABLE_DEFINITION' IN typenames THEN
    RETURN (has_values_space (expr\unary_generic_expression.operand));
  END_IF;
  IF 'COMPLEX_NUMBER_LITERAL' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'LOGICAL_LITERAL' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'BINARY_LITERAL' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'MATHS_ENUM_LITERAL' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'REAL_TUPLE_LITERAL' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'INTEGER_TUPLE_LITERAL' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'ATOM_BASED_LITERAL' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'MATHS_TUPLE_LITERAL' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'PARTIAL_DERIVATIVE_EXPRESSION' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  IF 'DEFINITE_INTEGRAL_EXPRESSION' IN typenames THEN
    RETURN (TRUE);
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- has_values_space
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

expr: (input) The expression to be tested.

return: (output) The BOOLEAN indication of whether or not the expression is recognized as a mathematical expression.

4.6.57 list_selected_components

This function constructs the list of selected components from an aggregate of **maths_tuple**.

NOTE This function is used in derivations and rules associated with the **finite_function** entity type.

EXPRESS specification:

```
*)
FUNCTION list_selected_components(aggr : AGGREGATE OF LIST OF maths_value;
                                  k    : positive_integer) : LIST OF maths_value;
  LOCAL
    result : LIST OF maths_value := [];
    j : INTEGER := 0;
  END_LOCAL;
  REPEAT i := LOINDEX (aggr) TO HIINDEX (aggr);
    IF k <= SIZEOF (aggr[i]) THEN
      INSERT (result, aggr[i][k], j);
      j := j + 1;
    END_IF;
  END_REPEAT;
  RETURN (result);
END_FUNCTION;  -- list_selected_components
(*
```

Argument definitions:

aggr: (input) The aggregate of tuples from which the selected components are to be extracted and listed.

k: (input) The ordinal position selected.

return: (output) The constructed list of selected components.

4.6.58 make_abstracted_expression_function

This function constructs an instance of **abstracted_expression_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_abstracted_expression_function(
                                   operands : LIST [2:?] OF generic_expression)
```


© ISO 2002 — All rights reserved

```
                                  : abstracted_expression_function;
  RETURN (abstracted_expression_function()
    || maths_function()
      || generic_expression()
    || quantifier_expression (remove_first (operands))  -- derived
      || multiple_arity_generic_expression (operands) );
END_FUNCTION;  -- make_abstracted_expression_function
(*
```

Argument definitions:

operands: (input) The list of operands, of which the first is the expression being abstracted to create the function and the rest are the variables, in order, to be treated as function inputs.

return: (output) The constructed complex entity instance of type **abstracted_expression_function**.

4.6.59 make_atom_based_literal

This function constructs an instance of **atom_based_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_atom_based_literal(lit_value : atom_based_value) : atom_based_literal;
  RETURN (atom_based_literal (lit_value)
    || generic_literal()
      || simple_generic_expression()
        || generic_expression() );
END_FUNCTION;  -- make_atom_based_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **atom_based_literal**.

4.6.60 make_b_spline_basis

This function constructs an instance of **b_spline_basis** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_b_spline_basis(degree         : nonnegative_integer;
                             repeated_knots : LIST [2:?] OF REAL) : b_spline_basis;
  RETURN (b_spline_basis (degree, repeated_knots)
    || maths_function()
```

© ISO 2002 — All rights reserved

```
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_b_spline_basis
(*
```

Argument definitions:

degree: (input) The degree of the B-spline basis functions.

repeated_knots: (input) The knot sequence for the basis functions.

return: (output) The constructed complex entity instance of **b_spline_basis**.

4.6.61 make_b_spline_function

This function constructs an instance of **b_spline_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_b_spline_function(coef  : maths_function;
                                bases : LIST [1:?] OF b_spline_basis)
                               : b_spline_function;
  RETURN (b_spline_function (bases)
    || maths_function()
      || generic_expression()
    || unary_generic_expression (coef) );
END_FUNCTION;  -- make_b_spline_function
(*
```

Argument definitions:

coef: (input) The coefficient table for the B-spline function.

bases: (input) The list of B-spline bases for the function.

return: (output) The constructed complex entity instance of **b_spline_function**.

4.6.62 make_banded_matrix

This function constructs an instance of **banded_matrix** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_banded_matrix(index_base    : zero_or_one;
                            shape         : LIST [1:?] OF positive_integer;
                            source        : maths_function;
                            first         : INTEGER;
```

 © ISO 2002 — All rights reserved

```
                            default_entry : maths_value;
                            below         : INTEGER;
                            above         : INTEGER;
                            order         : ordering_type) : banded_matrix;
  RETURN (banded_matrix (default_entry, below, above, order)
    || linearized_table_function (first)
      || explicit_table_function (index_base, shape)
        || maths_function()
          || generic_expression()
      || unary_generic_expression (source) );
END_FUNCTION;  -- make_banded_matrix
(*
```

Argument definitions:

index_base: (input) The base for the indices of the banded matrix.

shape: (input) The shape of the matrix as a list of positive integers.

source: (input) The 1D table function providing the non-default entries of the matrix.

first: (input) The location of the first matrix entry in **source**.

default_entry: (input) The value to use for all defaulted entries.

below: (input) The number of non-default diagonals below the main diagonal.

above: (input) The number of non-default diagonals above the main diagonal.

order: (input) The order of the entries in **source**.

return: (output) The constructed complex entity instance of **banded_matrix**.

4.6.63 make_basic_sparse_matrix

This function constructs an instance of **basic_sparse_matrix** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_basic_sparse_matrix(index_base    : zero_or_one;
                                  shape         : LIST [1:?] OF positive_integer;
                                  operands      : LIST [3:3] OF maths_function;
                                  default_entry : maths_value;
                                  order         : ordering_type)
                                 : basic_sparse_matrix;
  RETURN (basic_sparse_matrix (default_entry, order)
    || explicit_table_function (index_base, shape)
      || maths_function()
        || generic_expression()
    || multiple_arity_generic_expression (operands) );
END_FUNCTION;  -- make_basic_sparse_matrix
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

index_base: (input) The base for the indices of the sparse matrix.

shape: (input) The shape of the matrix as a list of positive integers.

operands: (input) The list of three 1D table functions providing the non-default entries of the matrix.

default_entry: (input) The value to use for all defaulted entries.

order: (input) The order of the entries encoded in **operands**.

return: (output) The constructed complex entity instance of **basic_sparse_matrix**.

4.6.64 make_binary_literal

This function constructs an instance of **binary_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_binary_literal(lit_value : BINARY) : binary_literal;
  RETURN (binary_literal (lit_value)
    || generic_literal()
      || simple_generic_expression()
        || generic_expression() );
END_FUNCTION;  -- make_binary_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **binary_literal**.

4.6.65 make_boolean_literal

This function constructs an instance of **boolean_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_boolean_literal(lit_value : BOOLEAN) : boolean_literal;
  RETURN (boolean_literal (lit_value)
    || simple_boolean_expression()
      || boolean_expression()
        || expression()
          || generic_expression()
      || simple_generic_expression()
    || generic_literal() );
```

 © ISO 2002 — All rights reserved

```
END_FUNCTION;  -- make_boolean_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **boolean_literal**.

4.6.66 make_cartesian_complex_number_region

This function constructs an instance of **cartesian_complex_number_region** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_cartesian_complex_number_region(real_constraint : real_interval;
                                              imag_constraint : real_interval)
                                             : cartesian_complex_number_region;
  RETURN (cartesian_complex_number_region (real_constraint, imag_constraint)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_cartesian_complex_number_region
(*
```

Argument definitions:

real_constraint: (input) The interval constraining the real parts.

imag_constraint: (input) The interval constraining the imaginary parts.

return: (output) The constructed complex entity instance of **cartesian_complex_number_region**.

4.6.67 make_complex_number_literal

This function constructs an instance of **complex_number_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_complex_number_literal(rpart, ipart : REAL) : complex_number_literal;
  RETURN (complex_number_literal (rpart, ipart)
    || generic_literal()
      || simple_generic_expression()
        || generic_expression() );
END_FUNCTION;  -- make_complex_number_literal
```

© ISO 2002 — All rights reserved

```
(*
```

Argument definitions:

rpart: (input) The real part of the complex number.

ipart: (input) The imaginary part of the complex number.

return: (output) The constructed complex entity instance of **complex_number_literal**.

4.6.68 make_constant_function

This function constructs an instance of **constant_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_constant_function(sole_value   : maths_value;
                                src_of_domn  : maths_space_or_function)
                               : constant_function;
  RETURN (constant_function (sole_value, src_of_domn)
    || maths_function()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_constant_function
(*
```

Argument definitions:

sole_value: (input) The value of the sole output of the constant function.

src_of_domn: (input) The source of the domain for the constant function.

return: (output) The constructed complex entity instance of **constant_function**.

4.6.69 make_cos_expression

This function constructs an instance of **cos_expression** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_cos_expression(operand : numeric_expression) : cos_expression;
  RETURN (cos_expression()
    || unary_numeric_call_expression()
      || unary_numeric_expression()
        || numeric_expression()
          || expression()
            || generic_expression()
```

 © ISO 2002 — All rights reserved

```
        || unary_generic_expression (operand) );
END_FUNCTION;  -- make_cos_expression
(*
```

Argument definitions:

operand: (input) The operand to which the cosine function is to be applied.

return: (output) The constructed complex entity instance of **cos_expression**.

4.6.70 make_definite_integral_expression

This function constructs an instance of **definite_integral_expression** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_definite_integral_expression(
                                   operands     : LIST [2:4] OF generic_expression;
                                   loinf, upinf : BOOLEAN)
                                          : definite_integral_expression;
  RETURN (definite_integral_expression (loinf, upinf)
    || quantifier_expression ([operands[2]])
      || multiple_arity_generic_expression (operands)
        || generic_expression() );
END_FUNCTION;  -- make_definite_integral_expression
(*
```

Argument definitions:

operands: (input) The two to four generic expressions representing the integrand expression, the variable of integration, and the finite lower and upper limits of integration when their presence is indicated.

loinf: (input) The BOOLEAN indicating whether the lower limit of integration is negative infinity, or present as an operand.

upinf: (input) The BOOLEAN indicating whether the upper limit of integration is positive infinity, or present as an operand.

return: (output) The constructed complex entity instance of **definite_integral_expression**.

4.6.71 make_definite_integral_function

This function constructs an instance of **definite_integral_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
```

© ISO 2002 — All rights reserved

```
FUNCTION make_definite_integral_function(integrand    : maths_function;
                                         varintg      : input_selector;
                                         loinf, upinf : BOOLEAN)
                                        : definite_integral_function;
  RETURN (definite_integral_function (varintg, loinf, upinf)
    || maths_function()
      || generic_expression()
    || unary_generic_expression (integrand) );
END_FUNCTION;  -- make_definite_integral_function
(*
```

Argument definitions:

integrand: (input) The function to be integrated.

varintg: (input) The integer identifying the variable of integration.

loinf: (input) The BOOLEAN indicating whether the lower limit of integration is negative infinity. or an input.

upinf: (input) The BOOLEAN indicating whether the upper limit of integration is positive infinity, or an input.

return: (output) The constructed complex entity instance of **definite_integral_function**.

4.6.72 make_elementary_function

This function constructs an instance of **elementary_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_elementary_function(func_id : elementary_function_enumerators)
                                 : elementary_function;
  RETURN (elementary_function (func_id)
    || maths_function()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_elementary_function
(*
```

Argument definitions:

func_id: (input) The enumeration value which identifies the function.

return: (output) The constructed complex entity instance of **elementary_function**.

4.6.73 make_elementary_space

This function constructs an instance of **elementary_space** given the necessary values for its explicit attributes.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION make_elementary_space(space_id : elementary_space_enumerators)
                              : elementary_space;
  RETURN (elementary_space (space_id)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_elementary_space
(*
```

Argument definitions:

space_id: (input) The enumeration value identifying the elementary space.

return: (output) The constructed complex entity instance of **elementary_space**.

4.6.74 make_environment

This function constructs an instance of **environment** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_environment(varbl : generic_variable;
                          sem   : variable_semantics) : environment;
  RETURN (environment (varbl, sem) );
END_FUNCTION;  -- make_environment
(*
```

Argument definitions:

varbl: (input) The variable to which semantics are to be associated.

src_of_domn: (input) The semantics to which a variable is to be associated.

return: (output) The constructed entity instance of **environment**.

4.6.75 make_expression_denoted_function

This function constructs an instance of **expression_denoted_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_expression_denoted_function(expression : generic_expression)
                                         : expression_denoted_function;
```

© ISO 2002 — All rights reserved

```
  RETURN (expression_denoted_function()
    || maths_function()
      || generic_expression()
    || unary_generic_expression (expression) );
END_FUNCTION;  -- make_expression_denoted_function
(*
```

Argument definitions:

functions: (input) The expression which denotes the function, that is, the expression whose value is being recognized as a mathematical function even though its type, as constructed, is not **maths_function**.

return: (output) The constructed complex entity instance of **expression_denoted_function**.

4.6.76 make_extended_tuple_space

This function constructs an instance of **extended_tuple_space** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_extended_tuple_space(base     : product_space;
                                   extender : maths_space) : extended_tuple_space;
  RETURN (extended_tuple_space (base, extender)
    || maths_space ()
      || generic_expression()
    || generic_literal ()
      || simple_generic_expression() );
END_FUNCTION;  -- make_extended_tuple_space
(*
```

Argument definitions:

factors: (input) The factor spaces for the product space.

return: (output) The constructed complex entity instance of **extended_tuple_space**.

4.6.77 make_finite_function

This function constructs an instance of **finite_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_finite_function(pairs : SET [1:?] OF LIST [2:2] OF maths_value)
                              : finite_function;
  RETURN (finite_function (pairs)
    || maths_function()
```

 © ISO 2002 — All rights reserved

```
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_finite_function
(*
```

Argument definitions:

pairs: (input) The set of ordered pairs which defines the function.

return: (output) The constructed complex entity instance of **finite_function**.

4.6.78 make_finite_integer_interval

This function constructs an instance of **finite_integer_interval** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_finite_integer_interval(min : INTEGER;
                                      max : INTEGER) : finite_integer_interval;
  RETURN (finite_integer_interval (min, max)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_finite_integer_interval
(*
```

Argument definitions:

min: (input) The least integer in the interval.

max: (input) The largest integer in the interval.

return: (output) The constructed complex entity instance of **finite_integer_interval**.

4.6.79 make_finite_real_interval

This function constructs an instance of **finite_real_interval** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_finite_real_interval(min    : REAL;
                                   minclo : open_closed;
                                   max    : REAL;
                                   maxclo : open_closed) : finite_real_interval;
  RETURN (finite_real_interval (min, minclo, max, maxclo)
```

© ISO 2002 — All rights reserved

```
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_finite_real_interval
(*
```

Argument definitions:

min: (input) The greatest lower bound for this interval.

minclo: (input) The indicator for whether the interval is closed or open at the lower end.

max: (input) The least upper bound for this interval.

maxclo: (input) The indicator for whether the interval is closed or open at the upper end.

return: (output) The constructed complex entity instance of **real_interval_from_min**.

4.6.80 make_finite_space

This function constructs an instance of **finite_space** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_finite_space(members : SET OF maths_value) : finite_space;
  RETURN (finite_space (members)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_finite_space
(*
```

Argument definitions:

members: (input) The set of members of the space to be constructed.

return: (output) The constructed complex entity instance of **finite_space**.

4.6.81 make_function_application

This function constructs an instance of **function_application** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_function_application(afunction : maths_function_select;
                                   arguments : LIST [1:?] OF maths_value)
```

 © ISO 2002 — All rights reserved

```
                                  : function_application;
  RETURN (function_application (afunction, arguments)
    || multiple_arity_generic_expression (convert_to_maths_function (afunction) +
      convert_to_operands (arguments))  -- derived
      || generic_expression() );
END_FUNCTION;  -- make_function_application
(*
```

Argument definitions:

afunction: (input) The function to be applied.

arguments: (input) The list of arguments to which the function is to be applied.

return: (output) The constructed complex entity instance of **function_application**.

4.6.82 make_function_space

This function constructs an instance of **function_space** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_function_space(domain_constraint : space_constraint_type;
                             domain_argument   : maths_space;
                             range_constraint  : space_constraint_type;
                             range_argument    : maths_space) : function_space;
  RETURN (function_space (domain_constraint, domain_argument, range_constraint,
    range_argument)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_function_space
(*
```

Argument definitions:

domain_constraint: (input) The enumeration value identifying the relationship of a member's domain space to the **domain_argument**.

domain_argument: (input) The space used in the domain constraint.

range_constraint: (input) The enumeration value identifying the relationship of a member's range space to the **range_argument**.

range_argument: (input) The space used in the range constraint.

return: (output) The constructed complex entity instance of **function_space**.

© ISO 2002 — All rights reserved

4.6.83 make_general_linear_function

This function constructs an instance of **general_linear_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_general_linear_function(mat       : maths_function;
                                      sum_index : one_or_two)
                                     : general_linear_function;
  RETURN (general_linear_function (sum_index)
    || maths_function()
      || generic_expression()
    || unary_generic_expression (mat) );
END_FUNCTION;  -- make_general_linear_function
(*
```

Argument definitions:

mat: (input) The 2D table function providing the coefficients.

sum_index: (input) Whether to sum on the first or second index.

return: (output) The constructed complex entity instance of **general_linear_function**.

4.6.84 make_int_literal

This function constructs an instance of **int_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_int_literal(lit_value : INTEGER) : int_literal;
  RETURN (int_literal ()
    || literal_number(lit_value)
      || simple_numeric_expression()
        || numeric_expression()
          || expression()
            || generic_expression()
        || simple_generic_expression()
      || generic_literal() );
END_FUNCTION;  -- make_int_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **int_literal**.

 © ISO 2002 — All rights reserved

4.6.85 make_integer_interval_from_min

This function constructs an instance of **integer_interval_from_min** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_integer_interval_from_min(min : INTEGER)
  : integer_interval_from_min;
  RETURN (integer_interval_from_min (min)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_integer_interval_from_min
(*
```

Argument definitions:

min: (input) The least integer in the interval.

return: (output) The constructed complex entity instance of **integer_interval_from_min**.

4.6.86 make_listed_complex_number_data

This function constructs an instance of **listed_complex_number_data** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_listed_complex_number_data(index_base : zero_or_one;
                                         values     : LIST [2:?] OF REAL)
                                        : listed_complex_number_data;
  RETURN (listed_complex_number_data (values)
    || explicit_table_function (index_base, [SIZEOF (values)/2])  -- 2nd derived
      || maths_function()
        || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_listed_complex_number_data
(*
```

Argument definitions:

index_base: (input) The low index for indexing this table.

values: (input) The list of pairs of real values representing the real and imaginary parts of the complex numbers to return for successive input values.

return: (output) The constructed complex entity instance of **listed_complex_number_data**.

© ISO 2002 — All rights reserved

4.6.87 make_listed_data

This function constructs an instance of **listed_data** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_listed_data(index_base  : zero_or_one;
                          values      : LIST [2:?] OF maths_value;
                          value_range : maths_space) : listed_data;
  RETURN (listed_data (values, value_range)
    || explicit_table_function (index_base, [SIZEOF (values)])  -- 2nd derived
      || maths_function()
        || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_listed_data
(*
```

Argument definitions:

index_base: (input) The low index for indexing this table.

values: (input) The list of values to return for successive index values.

value_range: (input) The space containing the members of **values**.

return: (output) The constructed complex entity instance of **listed_data**.

4.6.88 make_listed_integer_data

This function constructs an instance of **listed_integer_data** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_listed_integer_data(index_base : zero_or_one;
                                  values     : LIST [1:?] OF INTEGER)
                                 : listed_integer_data;
  RETURN (listed_integer_data (values)
    || explicit_table_function (index_base, [SIZEOF (values)])  -- 2nd derived
      || maths_function()
        || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_listed_integer_data
(*
```

 © ISO 2002 — All rights reserved

Argument definitions:

index_base: (input) The low index for indexing this table.

values: (input) The list of values to return for successive index values.

return: (output) The constructed complex entity instance of **listed_integer_data**.

4.6.89 make_listed_product_space

This function constructs an instance of **listed_product_space** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_listed_product_space(factors : LIST OF maths_space)
                                  : listed_product_space;
  RETURN (listed_product_space (factors)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_listed_product_space
(*
```

Argument definitions:

factors: (input) The factor spaces for the product space.

return: (output) The constructed complex entity instance of **listed_product_space**.

4.6.90 make_listed_real_data

This function constructs an instance of **listed_real_data** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_listed_real_data(index_base : zero_or_one;
                               values     : LIST [1:?] OF REAL)
                              : listed_real_data;
  RETURN (listed_real_data (values)
    || explicit_table_function (index_base, [SIZEOF (values)])  -- 2nd derived
      || maths_function()
        || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_listed_real_data
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

index_base: (input) The low index for indexing this table.

values: (input) The list of values to return for successive index values.

return: (output) The constructed complex entity instance of **listed_real_data**.

4.6.91 make_logical_literal

This function constructs an instance of **logical_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_logical_literal(lit_value : LOGICAL) : logical_literal;
  RETURN (logical_literal (lit_value)
    || generic_literal()
      || simple_generic_expression()
        || generic_expression() );
END_FUNCTION;  -- make_logical_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **logical_literal**.

4.6.92 make_maths_enum_literal

This function constructs an instance of **maths_enum_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_maths_enum_literal(lit_value : maths_enum_atom) : maths_enum_literal;
  RETURN (maths_enum_literal (lit_value)
    || generic_literal()
      || simple_generic_expression()
        || generic_expression() );
END_FUNCTION;  -- make_maths_enum_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **maths_enum_literal**.

 © ISO 2002 — All rights reserved

4.6.93 make_maths_real_variable

This function constructs an instance of **maths_real_variable** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_maths_real_variable(values_space : maths_space;
                                  name         : label) : maths_real_variable;
  RETURN (maths_real_variable()
    || maths_variable (values_space, name)
      || generic_variable()
        || simple_generic_expression()
          || generic_expression()
    || real_numeric_variable()
      || numeric_variable()
        || variable() );
END_FUNCTION;  -- make_maths_real_variable
(*
```

Argument definitions:

values_space: (input) The space over which the variable ranges.

name: (input) The name of the variable.

return: (output) The constructed complex entity instance of **maths_real_variable**.

4.6.94 make_maths_tuple_literal

This function constructs an instance of **maths_tuple_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_maths_tuple_literal(lit_value : LIST OF maths_value)
                                 : maths_tuple_literal;
  RETURN (maths_tuple_literal (lit_value)
    || generic_literal()
      || simple_generic_expression()
        || generic_expression() );
END_FUNCTION;  -- make_maths_tuple_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **maths_tuple_literal**.

© ISO 2002 — All rights reserved

4.6.95 make_mult_expression

This function constructs an instance of **mult_expression** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_mult_expression(operands : LIST [2:?] OF generic_expression)
                             : mult_expression;
  RETURN (mult_expression()
    || multiple_arity_numeric_expression()
      || numeric_expression()
        || expression()
          || generic_expression()
      || multiple_arity_generic_expression (operands) );
END_FUNCTION;  -- make_mult_expression
(*
```

Argument definitions:

operands: (input) The operands which are to be multiplied.

return: (output) The constructed complex entity instance of **mult_expression**.

4.6.96 make_parallel_composed_function

This function constructs an instance of **parallel_composed_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_parallel_composed_function(srcdom    : maths_space_or_function;
                                         prepfuncs : LIST [2:?] OF maths_function;
                                         finfunc   : maths_function_select)
                                        : parallel_composed_function;
  RETURN (parallel_composed_function (srcdom, prepfuncs, finfunc)
    || maths_function()
      || generic_expression()
    || multiple_arity_generic_expression (convert_to_operands_prcmfn (
      srcdom, prepfuncs, finfunc)) );  -- derived
END_FUNCTION;  -- make_parallel_composed_function
(*
```

Argument definitions:

srcdom: (input) The value of the **source_of_domain** attribute.

prepfuncs: (input) The list of values for the **prep_functions** attribute.

finfunc: (input) The value of the **final_function** attribute.

 © ISO 2002 — All rights reserved

return: (output) The constructed complex entity instance of **parallel_composed_function**.

4.6.97 make_partial_derivative_expression

This function constructs an instance of **partial_derivative_expression** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_partial_derivative_expression(derivand : generic_expression;
                                            dvars    : LIST [1:?] OF maths_variable;
                                            extend   : extension_options)
                                           : partial_derivative_expression;
  RETURN (partial_derivative_expression (dvars, extend)
    || unary_generic_expression (derivand)
      || generic_expression() );
END_FUNCTION;  -- make_partial_derivative_expression
(*
```

Argument definitions:

derivand: (input) The expression to be differentiated.

dvars: (input) The list of differentiation variables.

extend: (input) The enumeration value identifying the extension option.

return: (output) The constructed complex entity instance of **partial_derivative_expression**.

4.6.98 make_partial_derivative_function

This function constructs an instance of **partial_derivative_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_partial_derivative_function(derivand : maths_function;
                                          dvars    : LIST [1:?] OF input_selector;
                                          extend   : extension_options)
                                         : partial_derivative_function;
  RETURN (partial_derivative_function (dvars, extend)
    || maths_function()
      || generic_expression()
    || unary_generic_expression (derivand) );
END_FUNCTION;  -- make_partial_derivative_function
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

derivand: (input) The function to be differentiated.

dvars: (input) The list of integers identifying the differentiation variables.

extend: (input) The enumeration value identifying the extension option.

return: (output) The constructed complex entity instance of **partial_derivative_function**.

4.6.99 make_polar_complex_number_region

This function constructs an instance of **polar_complex_number_region** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_polar_complex_number_region(centre         : complex_number_literal;
                                          dis_constraint : real_interval;
                                          dir_constraint : finite_real_interval)
                                         : polar_complex_number_region;
  RETURN (polar_complex_number_region (centre, dis_constraint, dir_constraint)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_polar_complex_number_region
(*
```

Argument definitions:

centre: (input) The centre of the polar region.

dis_constraint: (input) The interval of distances from centre in the region.

dir_constraint: (input) The interval of directions from centre in the region.

return: (output) The constructed complex entity instance of **polar_complex_number_region**.

4.6.100 make_rationalize_function

This function constructs an instance of **rationalize_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_rationalize_function(fun : maths_function) : rationalize_function;
  RETURN (rationalize_function()
    || maths_function()
      || generic_expression()
```

 © ISO 2002 — All rights reserved

```
    || unary_generic_expression (fun) );
END_FUNCTION;  -- make_rationalize_function
(*
```

Argument definitions:

fun: (input) The tuple-producing function whose components are used to form the rational function.

return: (output) The constructed complex entity instance of **rationalize_function**.

4.6.101 make_real_interval_from_min

This function constructs an instance of **real_interval_from_min** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_real_interval_from_min(min    : REAL;
                                     minclo : open_closed) : real_interval_from_min;
  RETURN (real_interval_from_min (min, minclo)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_real_interval_from_min
(*
```

Argument definitions:

min: (input) The greatest lower bound for this interval.

minclo: (input) The indicator for whether the interval is closed or open at the lower end.

return: (output) The constructed complex entity instance of **real_interval_from_min**.

4.6.102 make_real_interval_to_max

This function constructs an instance of **real_interval_to_max** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_real_interval_to_max(max    : REAL;
                                   maxclo : open_closed) : real_interval_to_max;
  RETURN (real_interval_to_max (max, maxclo)
    || maths_space()
      || generic_expression()
    || generic_literal()
```

© ISO 2002 — All rights reserved

```
      || simple_generic_expression() );
END_FUNCTION;  -- make_real_interval_to_max
(*
```

Argument definitions:

max: (input) The least upper bound for this interval.

maxclo: (input) The indicator for whether the interval is closed or open at the upper end.

return: (output) The constructed complex entity instance of **real_interval_to_max**.

4.6.103 make_real_literal

This function constructs an instance of **real_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_real_literal(lit_value : REAL) : real_literal;
  RETURN (real_literal ()
    || literal_number(lit_value)
      || simple_numeric_expression()
        || numeric_expression()
          || expression()
            || generic_expression()
        || simple_generic_expression()
      || generic_literal() );
END_FUNCTION;  -- make_real_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **real_literal**.

4.6.104 make_regular_table_function

This function constructs an instance of **regular_table_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_regular_table_function(index_base : zero_or_one;
                                     shape      : LIST [1:?] OF positive_integer;
                                     operand    : maths_function;
                                     first      : INTEGER;
                                     increments : LIST [1:?] OF INTEGER)
```


© ISO 2002 — All rights reserved

```
                                   : regular_table_function;
  RETURN (regular_table_function (increments)
    || linearized_table_function (first)
      || explicit_table_function (index_base, shape)
        || maths_function()
          || generic_expression()
      || unary_generic_expression (operand) );
END_FUNCTION;  -- make_regular_table_function
(*
```

Argument definitions:

index_base: (input) The low index for indexing this table.

shape: (input) The list of numbers of subscripts in each indexing position.

operand: (input) The one-dimensional array function supplying the table values.

first: (input) The index of the first table entry in the **operand** array function.

increments: (input) The list of position deltas in **operand** for each index.

return: (output) The constructed complex entity instance of **regular_table_function**.

4.6.105 make_reindexed_array_function

This function constructs an instance of **reindexed_array_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_reindexed_array_function(func       : maths_function;
                                       start_idxs : LIST [1:?] OF INTEGER)
                                     : reindexed_array_function;
  RETURN (reindexed_array_function(start_idxs)
    || maths_function()
      || generic_expression()
    || unary_generic_expression (func) );
END_FUNCTION;  -- make_reindexed_array_function
(*
```

Argument definitions:

func: (input) The array function to be reindexed.

start_idxs: (input) The starting indices for the new array function.

return: (output) The constructed complex entity instance of **reindexed_array_function**.

4.6.106 make_repackaging_function

This function constructs an instance of **repackaging_function** given the necessary values for its explicit attributes.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION make_repackaging_function(operand         : maths_function;
                                   input_repack    : repackage_options;
                                   output_repack   : repackage_options;
                                   selected_output : nonnegative_integer)
                                  : repackaging_function;
  RETURN (repackaging_function (input_repack, output_repack, selected_output)
    || maths_function()
      || generic_expression()
    || unary_generic_expression (operand) );
END_FUNCTION;  -- make_repackaging_function
(*
```

Argument definitions:

operand: (input) The function to be repackaged.

input_repack: (input) The repackaging to be done on the input before applying **operand**.

output_repack: (input) The repackaging to be done on the output after applying **operand**.

selected_output: (input) The value for the **selected_output** attribute.

return: (output) The constructed complex entity instance of **repackaging_function**.

4.6.107 make_selector_function

This function constructs an instance of **selector_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_selector_function(selector    : input_selector;
                                src_of_domn : maths_space_or_function)
                               : selector_function;
  RETURN (selector_function (selector, src_of_domn)
    || maths_function()
       || generic_expression()
    || generic_literal()
       || simple_generic_expression() );
END_FUNCTION;  -- make_selector_function
(*
```

Argument definitions:

selector: (input) The input to be selected.

src_of_domn: (input) The source of the domain for the selector function.

return: (output) The constructed complex entity instance of **selector_function**.

 © ISO 2002 — All rights reserved

4.6.108 make_series_composed_function

This function constructs an instance of **series_composed_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_series_composed_function(functions : LIST [2:?] OF maths_function)
                                       : series_composed_function;
  RETURN (series_composed_function()
    || maths_function()
      || generic_expression()
    || multiple_arity_generic_expression (functions) );
END_FUNCTION;  -- make_series_composed_function
(*
```

Argument definitions:

functions: (input) The list of functions to be composed, in order of application.

return: (output) The constructed complex entity instance of **series_composed_function**.

4.6.109 make_sin_expression

This function constructs an instance of **sin_expression** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_sin_expression(operand : numeric_expression) : sin_expression;
  RETURN (sin_expression()
    || unary_numeric_call_expression()
      || unary_numeric_expression()
        || numeric_expression()
          || expression()
            || generic_expression()
        || unary_generic_expression (operand) );
END_FUNCTION;  -- make_sin_expression
(*
```

Argument definitions:

operand: (input) The operand to which the sine function is to be applied.

return: (output) The constructed complex entity instance of **sin_expression**.

© ISO 2002 — All rights reserved

4.6.110 make_standard_table_function

This function constructs an instance of **standard_table_function** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_standard_table_function(index_base : zero_or_one;
                                      shape      : LIST [1:?] OF positive_integer;
                                      operand    : maths_function;
                                      first      : INTEGER;
                                      order      : ordering_type)
                                     : standard_table_function;
  RETURN (standard_table_function (order)
    || linearized_table_function (first)
      || explicit_table_function (index_base, shape)
        || maths_function()
          || generic_expression()
      || unary_generic_expression (operand) );
END_FUNCTION;  -- make_standard_table_function
(*
```

Argument definitions:

index_base: (input) The low index for indexing this table.

shape: (input) The list of numbers of subscripts in each indexing position.

operand: (input) The one-dimensional array function supplying the table values.

first: (input) The index of the first table entry in the **operand** array function.

order: (input) The standard ordering (**by_rows** or **by_columns**) of the entries in the **operand** array function.

return: (output) The constructed complex entity instance of **standard_table_function**.

4.6.111 make_strict_triangular_matrix

This function constructs an instance of **strict_triangular_matrix** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_strict_triangular_matrix(index_base    : zero_or_one;
                                       shape         : LIST [1:?] OF positive_integer;
                                       source        : maths_function;
                                       first         : INTEGER;
                                       default_entry : maths_value;
                                       lo_up         : lower_upper;
                                       order         : ordering_type;
                                       main_diagonal_value : maths_value)
```

 © ISO 2002 — All rights reserved

```
                                       : strict_triangular_matrix;
  RETURN (strict_triangular_matrix (main_diagonal_value)
    || triangular_matrix (default_entry, lo_up, order)
      || linearized_table_function (first)
        || explicit_table_function (index_base, shape)
          || maths_function()
            || generic_expression()
        || unary_generic_expression (source) );
END_FUNCTION;  -- make_strict_triangular_matrix
(*
```

Argument definitions:

index_base: (input) The base for the indices of the strict_triangular matrix.

shape: (input) The shape of the matrix as a list of positive integers.

source: (input) The 1D table function providing the non-default entries of the matrix.

first: (input) The location of the first matrix entry in **source**.

default_entry: (input) The value to use for defaulted entries.

lo_up: (input) Whether the lower or upper triangle contains the non-default entries.

order: (input) The order of the entries in **source**.

main_diagonal_value: (input) The value to return for entries on the main diagonal.

return: (output) The constructed complex entity instance of **strict_triangular_matrix**.

4.6.112 make_string_literal

This function constructs an instance of **string_literal** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_string_literal(lit_value : STRING) : string_literal;
  RETURN (string_literal (lit_value)
    || simple_string_expression()
      || string_expression()
        || expression()
          || generic_expression()
      || simple_generic_expression()
    || generic_literal() );
END_FUNCTION;  -- make_string_literal
(*
```

Argument definitions:

lit_value: (input) The value for which a literal is to be constructed.

return: (output) The constructed complex entity instance of **string_literal**.

© ISO 2002 — All rights reserved

4.6.113 make_unary_minus_expression

This function constructs an instance of **unary_minus_expression** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_unary_minus_expression(operand : numeric_expression)
  : unary_minus_expression;
  RETURN (unary_minus_expression()
    || unary_numeric_call_expression()
      || unary_numeric_expression()
        || numeric_expression()
          || expression()
            || generic_expression()
        || unary_generic_expression (operand) );
END_FUNCTION;  -- make_unary_minus_expression
(*
```

Argument definitions:

operand: (input) The operand to which the unary negation is to be applied.

return: (output) The constructed complex entity instance of **unary_minus_expression**.

4.6.114 make_uniform_product_space

This function constructs an instance of **uniform_product_space** given the necessary values for its explicit attributes.

EXPRESS specification:

```
*)
FUNCTION make_uniform_product_space(base     : maths_space;
                                    exponent : positive_integer)
                                   : uniform_product_space;
  RETURN (uniform_product_space (base, exponent)
    || maths_space()
      || generic_expression()
    || generic_literal()
      || simple_generic_expression() );
END_FUNCTION;  -- make_uniform_product_space
(*
```

Argument definitions:

base: (input) The space used for all factors of the cartesian product.

exponent: (input) The number of factors to use.

return: (output) The constructed complex entity instance of **uniform_product_space**.

 © ISO 2002 — All rights reserved

4.6.115 max_exists

This function reports whether or not a maximum value exists for an interval. If applied to an instance of **maths_space** which is not an interval, FALSE is returned.

EXPRESS specification:

```
*)
FUNCTION max_exists(spc : maths_space) : BOOLEAN;
  LOCAL
    types : SET OF STRING := TYPEOF (spc);
  END_LOCAL;
  RETURN (bool(((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN types) OR
    ((schema_prefix + 'INTEGER_INTERVAL_TO_MAX') IN types) OR
    ((schema_prefix + 'FINITE_REAL_INTERVAL') IN types) OR
    ((schema_prefix + 'REAL_INTERVAL_TO_MAX') IN types)));
END_FUNCTION;  -- max_exists
(*
```

Argument definitions:

spc: (input) The space to be tested.

return: (output) A BOOLEAN value which is TRUE if the space is an interval which has an upper bound.

4.6.116 max_included

This function reports whether or not the least upper bound for an interval is a member of the interval. If applied to an instance of **maths_space** which is not an interval, or if the interval has no upper bound, FALSE is returned.

EXPRESS specification:

```
*)
FUNCTION max_included(spc : maths_space) : BOOLEAN;
  LOCAL
    types : SET OF STRING := TYPEOF (spc);
  END_LOCAL;
  IF ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN types) OR
    ((schema_prefix + 'INTEGER_INTERVAL_TO_MAX') IN types) THEN
    RETURN (TRUE);
  END_IF;
  IF ((schema_prefix + 'FINITE_REAL_INTERVAL') IN types) THEN
    RETURN (bool(spc\finite_real_interval.max_closure = closed));
  END_IF;
  IF ((schema_prefix + 'REAL_INTERVAL_TO_MAX') IN types) THEN
    RETURN (bool(spc\real_interval_to_max.max_closure = closed));
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- max_included
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

spc: (input) The space to be tested.

return: (output) A BOOLEAN value which is TRUE if the space is an interval which includes its least upper bound.

4.6.117 member_of

This function attempts to determine whether a given mathematical value is a member of a given mathematical space. Where the information accessible to this function is not sufficient to determine the matter, UNKNOWN is returned.

NOTE The EXPRESS code below introduces and uses the trivial function **fedex** solely to avoid an error in the NIST Fedex compiler. Although technically unnecessary, the extra code is correct and should have no detectable consequences.

EXPRESS specification:

```
*)
FUNCTION member_of(val : GENERIC:G;
                   spc : maths_space) : LOGICAL;

  -- Trivial function introduced to avoid NIST Fedex compiler error
  FUNCTION fedex(val : AGGREGATE OF GENERIC:X;
                 i   : INTEGER) : GENERIC:X;
    RETURN (val[i]);
  END_FUNCTION;  -- fedex

  LOCAL
    v : maths_value := simplify_maths_value (convert_to_maths_value (val));
    vtypes : SET OF STRING := stripped_typeof (v);
    s : maths_space := simplify_maths_space (spc);
    stypes : SET OF STRING := stripped_typeof (s);
    tmp_int : INTEGER;
    tmp_real : REAL;
    tmp_cmplx : complex_number_literal;
    lgcl, cum : LOGICAL;
    vspc, sspc : maths_space;
    smem : SET OF maths_value;
    factors : LIST OF maths_space;
  END_LOCAL;
  IF NOT EXISTS (s) THEN
    RETURN (FALSE);
  END_IF;
  IF NOT EXISTS (v) THEN
    RETURN (s = the_generics);
  END_IF;
  IF ('GENERIC_EXPRESSION' IN vtypes) AND
    NOT ('MATHS_SPACE' IN vtypes) AND
    NOT ('MATHS_FUNCTION' IN vtypes) AND
    NOT ('COMPLEX_NUMBER_LITERAL' IN vtypes) THEN
    IF has_values_space (v) THEN
      vspc := values_space_of (v);
      IF subspace_of (vspc, s) THEN
        RETURN (TRUE);
```

 © ISO 2002 — All rights reserved

```
    END_IF;
    IF NOT compatible_spaces (vspc, s) THEN
      RETURN (FALSE);
    END_IF;
    RETURN (UNKNOWN);
  END_IF;
  RETURN (UNKNOWN);
END_IF;
IF 'ELEMENTARY_SPACE' IN stypes THEN
  CASE s\elementary_space.space_id OF
  es_numbers :  RETURN (('NUMBER' IN vtypes) OR
                        ('COMPLEX_NUMBER_LITERAL' IN vtypes));
  es_complex_numbers :  RETURN ('COMPLEX_NUMBER_LITERAL' IN vtypes);
  es_reals :  RETURN (('REAL' IN vtypes) AND NOT ('INTEGER' IN vtypes));
  es_integers :  RETURN ('INTEGER' IN vtypes);
  es_logicals :  RETURN ('LOGICAL' IN vtypes);
  es_booleans :  RETURN ('BOOLEAN' IN vtypes);
  es_strings :  RETURN ('STRING' IN vtypes);
  es_binarys :  RETURN ('BINARY' IN vtypes);
  es_maths_spaces :  RETURN ('MATHS_SPACE' IN vtypes);
  es_maths_functions :  RETURN ('MATHS_FUNCTION' IN vtypes);
  es_generics :  RETURN (TRUE);
  END_CASE;
END_IF;
IF 'FINITE_INTEGER_INTERVAL' IN stypes THEN
  IF 'INTEGER' IN vtypes THEN
    tmp_int := v;
    RETURN ({s\finite_integer_interval.min <= tmp_int <=
      s\finite_integer_interval.max});
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'INTEGER_INTERVAL_FROM_MIN' IN stypes THEN
  IF 'INTEGER' IN vtypes THEN
    tmp_int := v;
    RETURN (s\integer_interval_from_min.min <= tmp_int);
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'INTEGER_INTERVAL_TO_MAX' IN stypes THEN
  IF 'INTEGER' IN vtypes THEN
    tmp_int := v;
    RETURN (tmp_int <= s\integer_interval_to_max.max);
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'FINITE_REAL_INTERVAL' IN stypes THEN
  IF ('REAL' IN vtypes) AND NOT ('INTEGER' IN vtypes) THEN
    tmp_real := v;
    IF s\finite_real_interval.min_closure = closed THEN
      IF s\finite_real_interval.max_closure = closed THEN
        RETURN ({s\finite_real_interval.min <= tmp_real <=
          s\finite_real_interval.max});
      ELSE
        RETURN ({s\finite_real_interval.min <= tmp_real <
          s\finite_real_interval.max});
      END_IF;
    ELSE
      IF s\finite_real_interval.max_closure = closed THEN
```

© ISO 2002 — All rights reserved

```
        RETURN ({s\finite_real_interval.min < tmp_real <=
          s\finite_real_interval.max});
      ELSE
        RETURN ({s\finite_real_interval.min < tmp_real <
          s\finite_real_interval.max});
      END_IF;
    END_IF;
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'REAL_INTERVAL_FROM_MIN' IN stypes THEN
  IF ('REAL' IN vtypes) AND NOT ('INTEGER' IN vtypes) THEN
    tmp_real := v;
    IF s\real_interval_from_min.min_closure = closed THEN
      RETURN (s\real_interval_from_min.min <= tmp_real);
    ELSE
      RETURN (s\real_interval_from_min.min < tmp_real);
   END_IF;
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'REAL_INTERVAL_TO_MAX' IN stypes THEN
  IF ('REAL' IN vtypes) AND NOT ('INTEGER' IN vtypes) THEN
    tmp_real := v;
    IF s\real_interval_to_max.max_closure = closed THEN
      RETURN (tmp_real <= s\real_interval_to_max.max);
    ELSE
      RETURN (tmp_real < s\real_interval_to_max.max);
   END_IF;
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'CARTESIAN_COMPLEX_NUMBER_REGION' IN stypes THEN
  IF 'COMPLEX_NUMBER_LITERAL' IN vtypes THEN
    RETURN (member_of(v\complex_number_literal.real_part,
      s\cartesian_complex_number_region.real_constraint) AND
      member_of(v\complex_number_literal.imag_part,
      s\cartesian_complex_number_region.imag_constraint));
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'POLAR_COMPLEX_NUMBER_REGION' IN stypes THEN
  IF 'COMPLEX_NUMBER_LITERAL' IN vtypes THEN
    tmp_cmplx := v;
    tmp_cmplx.real_part := tmp_cmplx.real_part -
      s\polar_complex_number_region.centre.real_part;
    tmp_cmplx.imag_part := tmp_cmplx.imag_part -
      s\polar_complex_number_region.centre.imag_part;
    tmp_real := SQRT (tmp_cmplx.real_part**2 + tmp_cmplx.imag_part**2);
    IF NOT member_of(tmp_real,
      s\polar_complex_number_region.distance_constraint) THEN
      RETURN (FALSE);
    END_IF;
    IF tmp_real = 0.0 THEN
      RETURN (TRUE);  -- The centre has no direction.
    END_IF;
    tmp_real := atan2(tmp_cmplx.imag_part,tmp_cmplx.real_part);
    RETURN (member_of(tmp_real,
      s\polar_complex_number_region.direction_constraint) OR
```

 © ISO 2002 — All rights reserved

```
        member_of(tmp_real + 2.0*PI,
        s\polar_complex_number_region.direction_constraint));
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'FINITE_SPACE' IN stypes THEN
    smem := s\finite_space.members;
    cum := FALSE;
    REPEAT i := 1 TO SIZEOF (smem);
      cum := cum OR equal_maths_values(v,smem[i]);
      IF cum = TRUE THEN
        RETURN (TRUE);
      END_IF;
    END_REPEAT;
    RETURN (cum);
  END_IF;
  IF 'UNIFORM_PRODUCT_SPACE' IN stypes THEN
    IF 'LIST' IN vtypes THEN
      IF SIZEOF (v) = s\uniform_product_space.exponent THEN
        sspc := s\uniform_product_space.base;
        cum := TRUE;
        REPEAT i := 1 TO SIZEOF (v);
          cum := cum AND member_of(v[i],sspc);
--          cum := cum AND member_of (fedex (v, i), sspc);
            -- See note above for explanation of fedex()
          IF cum = FALSE THEN
            RETURN (FALSE);
          END_IF;
        END_REPEAT;
        RETURN (cum);
      END_IF;
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'LISTED_PRODUCT_SPACE' IN stypes THEN
    IF 'LIST' IN vtypes THEN
      factors := s\listed_product_space.factors;
      IF SIZEOF (v) = SIZEOF (factors) THEN
        cum := TRUE;
        REPEAT i := 1 TO SIZEOF (v);
          cum := cum AND member_of(v[i],factors[i]);
--          cum := cum AND member_of (fedex (v, i), factors[i]);
            -- See note above for explanation of fedex()
          IF cum = FALSE THEN
            RETURN (FALSE);
          END_IF;
        END_REPEAT;
        RETURN (cum);
      END_IF;
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'EXTENDED_TUPLE_SPACE' IN stypes THEN
    IF 'LIST' IN vtypes THEN
      sspc := s\extended_tuple_space.base;
      tmp_int := space_dimension(sspc);
      IF SIZEOF (v) >= tmp_int THEN
        cum := TRUE;
        REPEAT i := 1 TO tmp_int;
```

© ISO 2002 — All rights reserved

```
          cum := cum AND member_of(v[i],factor_space(sspc,i));
--          cum := cum AND member_of (fedex (v, i), factor_space (sspc, i));
            -- See note above for explanation of fedex()
          IF cum = FALSE THEN
            RETURN (FALSE);
          END_IF;
        END_REPEAT;
        sspc := s\extended_tuple_space.extender;
        REPEAT i := tmp_int+1 TO SIZEOF (v);
          cum := cum AND member_of(v[i],sspc);
          IF cum = FALSE THEN
            RETURN (FALSE);
          END_IF;
        END_REPEAT;
        RETURN (cum);
      END_IF;
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'FUNCTION_SPACE' IN stypes THEN
    IF 'MATHS_FUNCTION' IN vtypes THEN
      vspc := v\maths_function.domain;
      sspc := s\function_space.domain_argument;
      CASE s\function_space.domain_constraint OF
      sc_equal : cum := equal_maths_spaces (vspc, sspc);
      sc_subspace : cum := subspace_of (vspc, sspc);
      sc_member : cum := member_of (vspc, sspc);
      END_CASE;
      IF cum = FALSE THEN
        RETURN (FALSE);
      END_IF;
      vspc := v\maths_function.range;
      sspc := s\function_space.range_argument;
      CASE s\function_space.range_constraint OF
      sc_equal : cum := cum AND equal_maths_spaces (vspc, sspc);
      sc_subspace : cum := cum AND subspace_of (vspc, sspc);
      sc_member : cum := cum AND member_of (vspc, sspc);
      END_CASE;
      RETURN (cum);
    END_IF;
    RETURN (FALSE);
  END_IF;
  -- Should be unreachable
  RETURN (UNKNOWN);
END_FUNCTION;  -- member_of
(*
```

Argument definitions:

val: (input) The value to be tested for membership in the space.

spc: (input) The space to be tested for containing the value.

return: (output) A LOGICAL value which is TRUE if the value can be determined to be a member of the space, FALSE if the value can be determined not to be a member of the space, and UNKNOWN, otherwise.

 © ISO 2002 — All rights reserved

4.6.118 min_exists

This function reports whether or not a minimum value exists for an interval. If applied to an instance of **maths_space** which is not an interval, FALSE is returned.

EXPRESS specification:

```
*)
FUNCTION min_exists(spc : maths_space) : BOOLEAN;
  LOCAL
    types : SET OF STRING := TYPEOF (spc);
  END_LOCAL;
  RETURN (bool(((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN types) OR
    ((schema_prefix + 'INTEGER_INTERVAL_FROM_MIN') IN types) OR
    ((schema_prefix + 'FINITE_REAL_INTERVAL') IN types) OR
    ((schema_prefix + 'REAL_INTERVAL_FROM_MIN') IN types)));
END_FUNCTION;  -- min_exists
(*
```

Argument definitions:

spc: (input) The space to be tested.

return: (output) A BOOLEAN value which is TRUE if the space is an interval which has an lower bound.

4.6.119 min_included

This function reports whether or not the greatest lower bound for an interval is a member of the interval. If applied to an instance of **maths_space** which is not an interval, or if the interval has no lower bound, FALSE is returned.

EXPRESS specification:

```
*)
FUNCTION min_included(spc : maths_space) : BOOLEAN;
  LOCAL
    types : SET OF STRING := TYPEOF (spc);
  END_LOCAL;
  IF ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN types) OR
    ((schema_prefix + 'INTEGER_INTERVAL_FROM_MIN') IN types) THEN
    RETURN (TRUE);
  END_IF;
  IF ((schema_prefix + 'FINITE_REAL_INTERVAL') IN types) THEN
    RETURN (bool(spc\finite_real_interval.min_closure = closed));
  END_IF;
  IF ((schema_prefix + 'REAL_INTERVAL_FROM_MIN') IN types) THEN
    RETURN (bool(spc\real_interval_from_min.min_closure = closed));
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- min_included
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

spc: (input) The space to be tested.

return: (output) A BOOLEAN value which is TRUE if the space is an interval which includes its greatest lower bound.

4.6.120 no_cyclic_domain_reference

This function verifies that the chain of references leading to a source for the domain of certain instances of **maths_function** terminates and does not lead to an endless loop.

NOTE In normal use, the second argument is initialized with the singleton set of the function whose **source_of_domain** attribute is to be checked. The presence of the second argument enables implementation by recursion.

EXPRESS specification:

```
*)
FUNCTION no_cyclic_domain_reference(ref  : maths_space_or_function;
                                    used : SET OF maths_function) : BOOLEAN;
  LOCAL
    typenames : SET OF STRING := TYPEOF (ref);
    func      : maths_function;
  END_LOCAL;
  IF (NOT EXISTS (ref)) OR (NOT EXISTS (used)) THEN
    RETURN (FALSE);
  END_IF;
  IF (schema_prefix + 'MATHS_SPACE') IN typenames THEN
    RETURN (TRUE);
  END_IF;
  func := ref;
  IF func IN used THEN
    RETURN (FALSE);
  END_IF;
  IF (schema_prefix + 'CONSTANT_FUNCTION') IN typenames THEN
    RETURN (no_cyclic_domain_reference (func\constant_function.source_of_domain,
      used + [func]));
  END_IF;
  IF (schema_prefix + 'SELECTOR_FUNCTION') IN typenames THEN
    RETURN (no_cyclic_domain_reference (func\selector_function.source_of_domain,
      used + [func]));
  END_IF;
  IF (schema_prefix + 'PARALLEL_COMPOSED_FUNCTION') IN typenames THEN
    RETURN (no_cyclic_domain_reference (
      func\parallel_composed_function.source_of_domain, used + [func]));
  END_IF;
  RETURN (TRUE);
END_FUNCTION;  -- no_cyclic_domain_reference
(*
```

Argument definitions:

ref: (input) The current reference.

 © ISO 2002 — All rights reserved

used: (input) The set of **maths_function** instances already referenced.

return: (output) A BOOLEAN value indicating whether the domain reference is proper.

4.6.121 no_cyclic_space_reference

This function returns true if the graph of **maths_space** references generated from its input **maths_space** contains no cycles and no members of its second argument.

NOTE In normal use, the second argument is initialized with the empty set. The presence of the second argument enables its implementation by recursion.

EXPRESS specification:

```
*)
FUNCTION no_cyclic_space_reference(spc  : maths_space;
                                   refs : SET OF maths_space) : BOOLEAN;
  LOCAL
    types : SET OF STRING;
    refs_plus : SET OF maths_space;
  END_LOCAL;
  IF (spc IN refs) THEN
    RETURN (FALSE);
  END_IF;
  types := TYPEOF (spc);
  refs_plus := refs + spc;
  IF (schema_prefix + 'FINITE_SPACE') IN types THEN
    RETURN (bool(SIZEOF (QUERY (sp <* QUERY (mem <* spc\finite_space.members |
      (schema_prefix + 'MATHS_SPACE') IN TYPEOF (mem)) |
      NOT no_cyclic_space_reference (sp, refs_plus))) = 0));
  END_IF;
  IF (schema_prefix + 'UNIFORM_PRODUCT_SPACE') IN types THEN
    RETURN (no_cyclic_space_reference (spc\uniform_product_space.base, refs_plus));
  END_IF;
  IF (schema_prefix + 'LISTED_PRODUCT_SPACE') IN types THEN
    RETURN (bool(SIZEOF (QUERY (fac <* spc\listed_product_space.factors |
      NOT no_cyclic_space_reference (fac, refs_plus))) = 0));
  END_IF;
  IF (schema_prefix + 'EXTENDED_TUPLE_SPACE') IN types THEN
    RETURN (no_cyclic_space_reference (spc\extended_tuple_space.base, refs_plus)
      AND no_cyclic_space_reference (spc\extended_tuple_space.extender, refs_plus));
  END_IF;
  -- spc contains no references to other spaces
  RETURN (TRUE);
END_FUNCTION;  -- no_cyclic_space_reference
(*
```

Argument definitions:

spc: (input) The space to be tested.

return: (output) A BOOLEAN value which is TRUE if the graph of space references reachable from the input has no cycles.

© ISO 2002 — All rights reserved

4.6.122 nondecreasing

This function determines whether or not a list of real values is in nondecreasing order.

EXPRESS specification:

```
*)
FUNCTION nondecreasing(lr : LIST OF REAL) : BOOLEAN;
  IF NOT EXISTS (lr) THEN
    RETURN (FALSE);
  END_IF;
  REPEAT j := 2 TO SIZEOF (lr);
    IF lr[j] < lr[j-1] THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;
(*
```

Argument definitions:

lr: (input) The list of real values to be tested.

return: (output) A BOOLEAN value which is TRUE if the members of the input list are in nondecreasing order.

4.6.123 number_superspace_of

This function returns the elementary numeric space corresponding to the type of the numbers which are members of the mathematical space represented by a **maths_space** instance. If the members of the mathematical space are not numbers, the indeterminate value is returned. The elementary space of all numbers is only returned for a finite space containing more than one kind of number among its members.

EXPRESS specification:

```
*)
FUNCTION number_superspace_of(spc : maths_space) : elementary_space;
  IF subspace_of_es(spc,es_integers) THEN  RETURN (the_integers);  END_IF;
  IF subspace_of_es(spc,es_reals)    THEN  RETURN (the_reals);     END_IF;
  IF subspace_of_es(spc,es_complex_numbers) THEN  RETURN (the_complex_numbers); END_IF;
  IF subspace_of_es(spc,es_numbers)  THEN  RETURN (the_numbers);   END_IF;
  RETURN (?);
END_FUNCTION;  -- number_superspace_of
(*
```

Argument definitions:

spc: (input) The **maths_space** value whose elementary numeric superspace is to be returned.

 © ISO 2002 — All rights reserved

return: (output) The elementary space of all numbers of the same type as the members of the input space, or, if the members are not all numbers, the indeterminate value (?).

4.6.124 number_tuple_subspace_check

This function identifies the **maths_space** values which represent subspaces of the number tuple spaces. In other words, this function identifies **maths_space** values representing mathematical spaces whose members are ordered tuples of numbers.

EXPRESS specification:

```
*)
FUNCTION number_tuple_subspace_check(spc : maths_space) : LOGICAL;
  LOCAL
    types : SET OF STRING := stripped_typeof(spc);
    factors : LIST OF maths_space;
    cum : LOGICAL := TRUE;
  END_LOCAL;
  IF 'UNIFORM_PRODUCT_SPACE' IN types THEN
    RETURN (subspace_of_es(spc\uniform_product_space.base,es_numbers));
  END_IF;
  IF 'LISTED_PRODUCT_SPACE' IN types THEN
    factors := spc\listed_product_space.factors;
    REPEAT i := 1 TO SIZEOF (factors);
      cum := cum AND subspace_of_es(factors[i],es_numbers);
    END_REPEAT;
    RETURN (cum);
  END_IF;
  IF 'EXTENDED_TUPLE_SPACE' IN types THEN
    cum := subspace_of_es(spc\extended_tuple_space.extender,es_numbers);
    cum := cum AND number_tuple_subspace_check(spc\extended_tuple_space.base);
    RETURN (cum);
  END_IF;
  RETURN (FALSE);
END_FUNCTION;
(*
```

Argument definitions:

spc: (input) The **maths_space** value to be checked.

return: (output) A LOGICAL value which is TRUE if it can be determined that the input space is one whose members are ordered tuples of numbers, FALSE if it can be determined the input space contains members which are not ordered tuples of numbers, and UNKNOWN, otherwise.

4.6.125 one_tuples_of

This function constructs the space of all one-tuples of elements from any given space.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION one_tuples_of(spc : maths_space) : tuple_space;
  RETURN (make_uniform_product_space (spc, 1));
END_FUNCTION;  -- one_tuples_of
(*
```

Argument definitions:

spc: (input) The space from which the elements used to form one-tuples are taken.

return: (output) The space of one-tuples of elements from the given space.

4.6.126 parallel_composed_function_composability_check

This function determines whether or not the associative Cartesian product of the ranges of the functions listed in its second argument is compatible with the domain of the function in its third argument.

EXPRESS specification:

```
*)
FUNCTION parallel_composed_function_composability_check(
                                     funcs  : LIST OF maths_function;
                                     final  : maths_function_select) : BOOLEAN;
  LOCAL
    tplsp : tuple_space := the_zero_tuple_space;
    finfun : maths_function := convert_to_maths_function (final);
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF (funcs);
    tplsp := assoc_product_space (tplsp, funcs[i].range);
  END_REPEAT;
  RETURN (compatible_spaces (tplsp, finfun.domain));
END_FUNCTION;  -- parallel_composed_function_composability_check
(*
```

Argument definitions:

funcs: (input) The list of functions whose ranges are to be used.

final: (input) The function whose domain is to be used.

return: (output) The BOOLEAN value signifying whether the associative Cartesian product of the ranges of functions in **funcs** are compatible with the domain of **final** or not.

4.6.127 parallel_composed_function_domain_check

This function determines whether or not all the domains of the list of functions in its second argument are compatible with its first argument.

 © ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION parallel_composed_function_domain_check(
                                comdom : tuple_space;
                                funcs  : LIST OF maths_function) : BOOLEAN;
  REPEAT i := 1 TO SIZEOF (funcs);
    IF NOT (compatible_spaces (comdom, funcs[i].domain)) THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- parallel_composed_function_domain_check
(*
```

Argument definitions:

comdom: (input) The space with which the function domains are checked for compatibility.

funcs: (input) The list of preparatory functions whose domains are to be checked for compatibility with **comdom**.

return: (output) The BOOLEAN value signifying whether all the domains of the functions in **funcs** are compatible with the value of **comdom** or not.

4.6.128 parse_express_identifier

This function locates the next position in a given string, starting from a given position, which cannot belong to a syntactically correct EXPRESS identifier. That is, the substring of the given string starting with the given position and ending at the position before the returned value of this function is either a null string, or begins with a letter and contains only letters, digits and underscores. If the starting position is beyond the end of the string, or does not contain a letter, the starting position is returned.

EXPRESS specification:

```
*)
FUNCTION parse_express_identifier(s : STRING;
                                  i : positive_integer) : positive_integer;
  LOCAL
    k : positive_integer;
  END_LOCAL;
  k := i;
  IF i <= LENGTH (s) THEN
    IF (s[i] LIKE '@') THEN
      REPEAT UNTIL (k > LENGTH (s)) OR
        ((s[k] <> '_') AND NOT (s[k] LIKE '@') AND NOT (s[k] LIKE '#'));
        k := k + 1;
      END_REPEAT;
    END_IF;
  END_IF;
  RETURN (k);
END_FUNCTION;  -- parse_express_identifier
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

s: (input) The string to be searched.

i: (input) The character position at which to begin the search.

return: (output) The next string position which cannot belong to an EXPRESS identifier.

4.6.129 partial_derivative_check

This function verifies that its inputs permit a meaningful partial derivative function. Specifically, it verifies that the members of **d_vars** correspond to components of **domain** and that those components are subspaces of the reals or the complex numbers.

EXPRESS specification:

```
*)
FUNCTION partial_derivative_check(domain : tuple_space;
                                  d_vars : LIST [1:?] OF input_selector) : BOOLEAN;
  LOCAL
    domn : tuple_space := domain;
    fspc : maths_space;
    dim : INTEGER;
    k : INTEGER;
  END_LOCAL;
  IF (space_dimension (domain) = 1) AND ((schema_prefix + 'TUPLE_SPACE') IN
    TYPEOF (factor1 (domain))) THEN
    domn := factor1 (domain);
  END_IF;
  dim := space_dimension (domn);
  REPEAT i := 1 TO SIZEOF (d_vars);
    k := d_vars[i];
    IF k > dim THEN
       RETURN (FALSE);
    END_IF;
    fspc := factor_space (domn, k);
    IF (NOT subspace_of_es (fspc,es_reals)) AND
      (NOT subspace_of_es (fspc,es_complex_numbers)) THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;  -- partial_derivative_check
(*
```

Argument definitions:

domain: (input) The domain of the function to be differentiated.

d_vars: (input) The values of type **input_selector** corresponding to implicit input variables with respect to which partial differentiation is to be performed.

return: (output) A BOOLEAN value indicating whether or not the inputs permit a meaningful partial derivative function.

 © ISO 2002 — All rights reserved

4.6.130 real_max

This function returns the least upper bound for an interval that has one. If applied to an instance of **maths_space** which is not an interval, or if the interval has no upper bound, the indeterminate value is returned.

EXPRESS specification:

```
*)
FUNCTION real_max(spc : maths_space) : REAL;
  LOCAL
    types : SET OF STRING := TYPEOF (spc);
  END_LOCAL;
  IF ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN types) THEN
    RETURN (spc\finite_integer_interval.max);
  END_IF;
  IF ((schema_prefix + 'INTEGER_INTERVAL_TO_MAX') IN types) THEN
    RETURN (spc\integer_interval_to_max.max);
  END_IF;
  IF ((schema_prefix + 'FINITE_REAL_INTERVAL') IN types) THEN
    RETURN (spc\finite_real_interval.max);
  END_IF;
  IF ((schema_prefix + 'REAL_INTERVAL_TO_MAX') IN types) THEN
    RETURN (spc\real_interval_to_max.max);
  END_IF;
  RETURN (?);
END_FUNCTION;  -- real_max
(*
```

Argument definitions:

spc: (input) The interval for which the least upper bound is to be obtained.

return: (output) The least upper bound as a real number, if one exists. Otherwise, the indeterminate value is returned.

4.6.131 real_min

This function returns the greatest lower bound for an interval that has one. If applied to an instance of **maths_space** which is not an interval, or if the interval has no lower bound, the indeterminate value is returned.

EXPRESS specification:

```
*)
FUNCTION real_min(spc : maths_space) : REAL;
  LOCAL
    types : SET OF STRING := TYPEOF (spc);
  END_LOCAL;
  IF ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN types) THEN
    RETURN (spc\finite_integer_interval.min);
  END_IF;
```

© ISO 2002 — All rights reserved

```
  IF ((schema_prefix + 'INTEGER_INTERVAL_FROM_MIN') IN types) THEN
    RETURN (spc\integer_interval_from_min.min);
  END_IF;
  IF ((schema_prefix + 'FINITE_REAL_INTERVAL') IN types) THEN
    RETURN (spc\finite_real_interval.min);
  END_IF;
  IF ((schema_prefix + 'REAL_INTERVAL_FROM_MIN') IN types) THEN
    RETURN (spc\real_interval_from_min.min);
  END_IF;
  RETURN (?);
END_FUNCTION;  -- real_min
(*
```

Argument definitions:

spc: (input) The interval for which the greatest lower bound is to be obtained.

return: (output) The greatest lower bound as a real number, if one exists. Otherwise, the indeterminate value is returned.

4.6.132 regular_indexing

This function computes the single index associated with a subscript tuple in the context of a subscript range base value, a table shape tuple, the tuple of increments for each subscript position, and the index of the first element. This function describes the indexing associated with an instance of **regular_table_function**.

EXPRESS specification:

```
*)
FUNCTION regular_indexing(sub   : LIST OF INTEGER;
                          base  : zero_or_one;
                          shape : LIST [1:?] OF positive_integer;
                          inc   : LIST [1:?] OF INTEGER;
                          first : INTEGER) : INTEGER;
  LOCAL
    k : INTEGER;
    index : INTEGER;
  END_LOCAL;
  IF NOT EXISTS (sub) OR NOT EXISTS (base) OR NOT EXISTS (shape) OR
    NOT EXISTS (inc) OR NOT EXISTS (first) THEN
    RETURN (?);
  END_IF;
  IF (SIZEOF (sub) <> SIZEOF (inc)) OR (SIZEOF (sub) <> SIZEOF (shape)) THEN
    RETURN (?);
  END_IF;
  index := first;
  REPEAT j := 1 TO SIZEOF (sub);
    IF NOT EXISTS (sub[j]) OR NOT EXISTS (inc[j]) THEN
      RETURN (?);
    END_IF;
    k := sub[j] - base;
    IF NOT ({0 <= k < shape[j]}) THEN
      RETURN (?);
    END_IF;
```

 © ISO 2002 — All rights reserved

```
    index := index + k*inc[j];
  END_REPEAT;
  RETURN (index);
END_FUNCTION;
(*
```

Argument definitions:

sub: (input) The subscript tuple for which a single index is to be computed.

base: (input) The base of the subscript ranges.

shape: (input) The shape of the table.

inc: (input) The increments associated with each subscript position.

first: (input) The index corresponding to the first subscript position.

return: (output) The single index corresponding to the input subscript tuple.

4.6.133 remove_first

This function takes any list and returns a copy with the first element, if any, removed.

EXPRESS specification:

```
*)
FUNCTION remove_first(alist : LIST OF GENERIC:GEN) : LIST OF GENERIC:GEN;
  LOCAL
    blist : LIST OF GENERIC:GEN := alist;
  END_LOCAL;
  IF SIZEOF (blist) > 0 THEN
    REMOVE (blist, 1);
  END_IF;
  RETURN (blist);
END_FUNCTION;  -- remove_first
(*
```

Argument definitions:

alist: (input) The input list.

return: (output) A copy of the input list with the first element removed.

4.6.134 repackage

This function transforms the input tuple space in accordance with the **repackage_option** value specified.

NOTE If called to unwrap a tuple space whose first factor space is not a tuple space, the algorithm may fail to produce an output of type **tuple_space**. It is assumed that the indeterminate value (?) is produced instead.

© ISO 2002 — All rights reserved

EXPRESS specification:

```
*)
FUNCTION repackage(tspace : tuple_space;
                   repckg : repackage_options) : tuple_space;
  CASE repckg OF
  ro_nochange : RETURN (tspace);
  ro_wrap_as_tuple : RETURN (one_tuples_of (tspace));
  ro_unwrap_tuple : RETURN (factor1 (tspace));
  OTHERWISE : RETURN (?);
  END_CASE;
END_FUNCTION;  -- repackage
(*
```

Argument definitions:

tspace: (input) The input tuple space.

repckg: (input) The repackage option specified.

return: (output) The resulting tuple space.

4.6.135 shape_of_array

This function returns a tuple of positive integers reflecting the numbers of subscripts in each indexing position for an array function. If the function is not an array function, the indeterminate value is returned.

EXPRESS specification:

```
*)
FUNCTION shape_of_array(func : maths_function) : LIST OF positive_integer;
  LOCAL
    tspace : tuple_space;
    temp : maths_space;
    result : LIST OF positive_integer := [];
  END_LOCAL;
  IF (schema_prefix + 'EXPLICIT_TABLE_FUNCTION') IN TYPEOF (func) THEN
    RETURN (func\explicit_table_function.shape);
  END_IF;
  tspace := func.domain;
  IF (space_dimension (tspace) = 1) AND ((schema_prefix + 'TUPLE_SPACE') IN
    TYPEOF (factor1 (tspace))) THEN
    tspace := factor1 (tspace);
  END_IF;
  REPEAT i := 1 TO space_dimension (tspace);
    temp := factor_space (tspace, i);
    IF NOT ((schema_prefix + 'FINITE_INTEGER_INTERVAL') IN TYPEOF (temp)) THEN
      RETURN (?);
    END_IF;
    INSERT (result, temp\finite_integer_interval.size, i-1);
  END_REPEAT;
  RETURN (result);
END_FUNCTION;  -- shape_of_array
(*
```

 © ISO 2002 — All rights reserved

Argument definitions:

func: (input) The input array function.

return: (output) The tuple of positive integers describing the shape of the array function.

4.6.136 simplify_function_application

This function attempts to simplify a **function_application** expression to obtain a simpler **maths_value**. When no simpler expression can be identified, the input expression may be returned unchanged.

EXPRESS specification:

```
*)
FUNCTION simplify_function_application(expr : function_application) : maths_value;
  FUNCTION ctmv(x : GENERIC:G) : maths_value;
    RETURN (convert_to_maths_value(x));
  END_FUNCTION;  -- local abbreviation for convert_to_maths_value function
  PROCEDURE parts(       c : complex_number_literal;
                  VAR x, y : REAL);
    x := c.real_part;  y := c.imag_part;
  END_PROCEDURE;  -- parts
  FUNCTION makec(x, y : REAL) : complex_number_literal;
    RETURN (make_complex_number_literal(x,y));
  END_FUNCTION;  -- local abbreviation for make_complex_number_literal function
  FUNCTION good_t(v  : maths_value;
                  tn : STRING) : BOOLEAN;
    LOCAL
      tpl : LIST OF maths_value;
    END_LOCAL;
    IF 'LIST' IN TYPEOF (v) THEN
      tpl := v;
      REPEAT i := 1 TO SIZEOF (tpl);
        IF NOT (tn IN TYPEOF (tpl[i])) THEN  RETURN (FALSE);  END_IF;
      END_REPEAT;
      RETURN (TRUE);
    END_IF;
    RETURN (FALSE);
  END_FUNCTION;  -- good_t
  CONSTANT
    cnlit : STRING := schema_prefix + 'COMPLEX_NUMBER_LITERAL';
  END_CONSTANT;
  LOCAL
    types : SET OF STRING := stripped_typeof(expr.func);
    ef_val : elementary_function_enumerators;
    is_elementary : BOOLEAN := FALSE;
    v, v1, v2, v3 : maths_value;
    vlist : LIST OF maths_value := [];
    gexpr : generic_expression;
    pairs : SET [1:?] OF LIST [2:2] OF maths_value;
    boo : BOOLEAN;
    lgc, cum : LOGICAL;
    j, k, n : INTEGER;
    p, q, r, s, t, u : REAL;
    str, st2 : STRING;
```

© ISO 2002 — All rights reserved

```
    bin, bi2 : BINARY;
    tpl, tp2 : LIST OF maths_value;
    mem :SET OF maths_value := [];
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF (expr.arguments);
    v := simplify_maths_value(expr.arguments[i]);
    INSERT (vlist, v, i-1);
  END_REPEAT;
  IF SIZEOF (vlist) >= 1 THEN  v1 := vlist[1];  END_IF;
  IF SIZEOF (vlist) >= 2 THEN  v2 := vlist[2];  END_IF;
  IF SIZEOF (vlist) >= 3 THEN  v3 := vlist[3];  END_IF;
  IF 'ELEMENTARY_FUNCTION_ENUMERATORS' IN types THEN
    ef_val := expr.func;
    is_elementary := TRUE;
  END_IF;
  IF 'ELEMENTARY_FUNCTION' IN types THEN
    ef_val := expr.func\elementary_function.func_id;
    is_elementary := TRUE;
  END_IF;
  IF is_elementary THEN
    CASE ef_val OF
    ef_and : BEGIN
      cum := TRUE;
      REPEAT i := SIZEOF (vlist) TO 1 BY -1;
        IF 'LOGICAL' IN TYPEOF (vlist[i]) THEN
          lgc := vlist[i];  cum := cum AND lgc;
          IF lgc = FALSE THEN  RETURN (ctmv(FALSE));  END_IF;
          REMOVE (vlist, i);
        END_IF;
      END_REPEAT;
      IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(cum));  END_IF;
      IF cum <> TRUE THEN  INSERT (vlist, ctmv(cum), 0);  END_IF;
      IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
      END;
    ef_or : BEGIN
      cum := FALSE;
      REPEAT i := SIZEOF (vlist) TO 1 BY -1;
        IF 'LOGICAL' IN TYPEOF (vlist[i]) THEN
          lgc := vlist[i];  cum := cum OR lgc;
          IF lgc = TRUE THEN  RETURN (ctmv(TRUE));  END_IF;
          REMOVE (vlist, i);
        END_IF;
      END_REPEAT;
      IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(cum));  END_IF;
      IF cum <> FALSE THEN  INSERT (vlist, ctmv(cum), 0);  END_IF;
      IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
      END;
    ef_not :
      IF 'LOGICAL' IN TYPEOF (v1) THEN  lgc := v1;  RETURN (ctmv(NOT lgc));  END_IF;
    ef_xor : BEGIN
      IF 'LOGICAL' IN TYPEOF (v1) THEN
        lgc := v1;
        IF 'LOGICAL' IN TYPEOF (v2) THEN  cum := v2;  RETURN (ctmv(lgc XOR cum));
        ELSE IF lgc = FALSE THEN  RETURN (ctmv(v2));
        ELSE IF lgc = UNKNOWN THEN  RETURN (ctmv(UNKNOWN));
        ELSE  RETURN (make_function_application(ef_not,[v2]));
        END_IF;  END_IF;  END_IF;
      ELSE IF 'LOGICAL' IN TYPEOF (v2) THEN
        lgc := v2;
```

 © ISO 2002 — All rights reserved

```
      IF lgc = FALSE THEN  RETURN (ctmv(v1));
      ELSE IF lgc = UNKNOWN THEN  RETURN (ctmv(UNKNOWN));
      ELSE  RETURN (make_function_application(ef_not,[v1]));
      END_IF;  END_IF;
    END_IF;  END_IF;
    END;
  ef_negate_i :
    IF 'INTEGER' IN TYPEOF (v1) THEN  j := v1;  RETURN (ctmv(-j));  END_IF;
  ef_add_i : BEGIN
    j := 0;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'INTEGER' IN TYPEOF (vlist[i]) THEN
        k := vlist[i];  j := j + k;
        REMOVE (vlist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(j));  END_IF;
    IF j <> 0 THEN  INSERT (vlist, ctmv(j), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_subtract_i :
    IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      j := v1;  k := v2;  RETURN (ctmv(j - k));
    END_IF;
  ef_multiply_i : BEGIN
    j := 1;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'INTEGER' IN TYPEOF (vlist[i]) THEN
        k := vlist[i];  j := j * k;
        REMOVE (vlist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(j));  END_IF;
    IF j <> 1 THEN  INSERT (vlist, ctmv(j), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_divide_i :
    IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      j := v1;  k := v2;  RETURN (ctmv(j DIV k));
    END_IF;
  ef_mod_i :
    IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      j := v1;  k := v2;  RETURN (ctmv(j MOD k));
    END_IF;
  ef_exponentiate_i :
    IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      j := v1;  k := v2;  n := 1;
      REPEAT i := 1 TO ABS(k);  n := n * j;  END_REPEAT;
      IF k < 0 THEN  n := 1 DIV n;  END_IF;
      RETURN (ctmv(n));
    END_IF;
  ef_eq_i :
    IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      j := v1;  k := v2;  RETURN (ctmv(j = k));
    END_IF;
  ef_ne_i :
    IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      j := v1;  k := v2;  RETURN (ctmv(j <> k));
    END_IF;
```

© ISO 2002 — All rights reserved

```
ef_gt_i :
  IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
    j := v1;  k := v2;  RETURN (ctmv(j > k));
  END_IF;
ef_lt_i :
  IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
    j := v1;  k := v2;  RETURN (ctmv(j < k));
  END_IF;
ef_ge_i :
  IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
    j := v1;  k := v2;  RETURN (ctmv(j >= k));
  END_IF;
ef_le_i :
  IF ('INTEGER' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
    j := v1;  k := v2;  RETURN (ctmv(j <= k));
  END_IF;
ef_abs_i :
  IF 'INTEGER' IN TYPEOF (v1) THEN  j := v1;  RETURN (ctmv(ABS(j)));  END_IF;
ef_max_i : BEGIN
  boo := FALSE;
  REPEAT i := SIZEOF (vlist) TO 1 BY -1;
    IF 'INTEGER' IN TYPEOF (vlist[i]) THEN
      IF boo THEN  k := vlist[i];  IF k > j THEN  j := k;  END_IF;
      ELSE  j := vlist[i];  boo := TRUE;  END_IF;
      REMOVE (vlist, i);
    END_IF;
  END_REPEAT;
  IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(j));  END_IF;
  IF boo THEN  INSERT (vlist, ctmv(j), 0);  END_IF;
  IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
  END;
ef_min_i : BEGIN
  boo := FALSE;
  REPEAT i := SIZEOF (vlist) TO 1 BY -1;
    IF 'INTEGER' IN TYPEOF (vlist[i]) THEN
      IF boo THEN  k := vlist[i];  IF k < j THEN  j := k;  END_IF;
      ELSE  j := vlist[i];  boo := TRUE;  END_IF;
      REMOVE (vlist, i);
    END_IF;
  END_REPEAT;
  IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(j));  END_IF;
  IF boo THEN  INSERT (vlist, ctmv(j), 0);  END_IF;
  IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
  END;
-- ef_if_i : combined with ef_if
ef_negate_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(-r));  END_IF;
ef_reciprocal_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(1.0/r));  END_IF;
ef_add_r : BEGIN
  r := 0.0;
  REPEAT i := SIZEOF (vlist) TO 1 BY -1;
    IF 'REAL' IN TYPEOF (vlist[i]) THEN
      s := vlist[i];  r := r + s;
      REMOVE (vlist, i);
    END_IF;
  END_REPEAT;
  IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(r));  END_IF;
  IF r <> 0.0 THEN  INSERT (vlist, ctmv(r), 0);  END_IF;
```

 © ISO 2002 — All rights reserved

```
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_subtract_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  RETURN (ctmv(r - s));
    END_IF;
  ef_multiply_r : BEGIN
    r := 1.0;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'REAL' IN TYPEOF (vlist[i]) THEN
        s := vlist[i];  r := r * s;
        REMOVE (vlist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(r));  END_IF;
    IF r <> 1.0 THEN  INSERT (vlist, ctmv(r), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_divide_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  RETURN (ctmv(r / s));
    END_IF;
  ef_mod_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  t := r/s;  j := t DIV 1;
      IF (t < 0.0) AND (j <> t) THEN  j := j - 1;  END_IF;
      RETURN (ctmv(r - j * s));
    END_IF;
  ef_exponentiate_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  RETURN (ctmv(r ** s));
    END_IF;
  ef_exponentiate_ri :
    IF ('REAL' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      r := v1;  k := v2;  t := 1.0;
      REPEAT i := 1 TO ABS(k);  t := t * r;  END_REPEAT;
      IF k < 0 THEN  t := 1.0/t;  END_IF;
      RETURN (ctmv(t));
    END_IF;
  ef_eq_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  RETURN (ctmv(r = s));
    END_IF;
  ef_ne_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  RETURN (ctmv(r <> s));
    END_IF;
  ef_gt_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  RETURN (ctmv(r > s));
    END_IF;
  ef_lt_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  RETURN (ctmv(r < s));
    END_IF;
  ef_ge_r :
    IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
      r := v1;  s := v2;  RETURN (ctmv(r >= s));
    END_IF;
```

© ISO 2002 — All rights reserved

```
ef_le_r :
  IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
    r := v1;  s := v2;  RETURN (ctmv(r <= s));
  END_IF;
ef_abs_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(ABS(r)));  END_IF;
ef_max_r : BEGIN
  boo := FALSE;
  REPEAT i := SIZEOF (vlist) TO 1 BY -1;
    IF 'REAL' IN TYPEOF (vlist[i]) THEN
      IF boo THEN  s := vlist[i];  IF s > r THEN  r := s;  END_IF;
      ELSE  r := vlist[i];  boo := TRUE;  END_IF;
      REMOVE (vlist, i);
    END_IF;
  END_REPEAT;
  IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(r));  END_IF;
  IF boo THEN  INSERT (vlist, ctmv(r), 0);  END_IF;
  IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
  END;
ef_min_r : BEGIN
  boo := FALSE;
  REPEAT i := SIZEOF (vlist) TO 1 BY -1;
    IF 'REAL' IN TYPEOF (vlist[i]) THEN
      IF boo THEN  s := vlist[i];  IF s < r THEN  r := s;  END_IF;
      ELSE  r := vlist[i];  boo := TRUE;  END_IF;
      REMOVE (vlist, i);
    END_IF;
  END_REPEAT;
  IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(r));  END_IF;
  IF boo THEN  INSERT (vlist, ctmv(r), 0);  END_IF;
  IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
  END;
ef_acos_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(ACOS(r)));  END_IF;
ef_asin_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(ASIN(r)));  END_IF;
ef_atan2_r :
  IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
    r := v1;  s := v2;  RETURN (ctmv(atan2(r,s)));
  END_IF;
ef_cos_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(COS(r)));  END_IF;
ef_exp_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(EXP(r)));  END_IF;
ef_ln_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(LOG(r)));  END_IF;
ef_log2_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(LOG2(r)));  END_IF;
ef_log10_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(LOG10(r)));  END_IF;
ef_sin_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(SIN(r)));  END_IF;
ef_sqrt_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(SQRT(r)));  END_IF;
ef_tan_r :
  IF 'REAL' IN TYPEOF (v1) THEN  r := v1;  RETURN (ctmv(TAN(r)));  END_IF;
-- ef_if_r : combined with ef_if
ef_form_c :
  IF ('REAL' IN TYPEOF (v1)) AND ('REAL' IN TYPEOF (v2)) THEN
```

 © ISO 2002 — All rights reserved

```
      r := v1;  s := v2;  RETURN (makec(r,s));
    END_IF;
  ef_rpart_c :
    IF cnlit IN TYPEOF (v1) THEN
      RETURN (ctmv(v1\complex_number_literal.real_part));
    END_IF;
  ef_ipart_c :
    IF cnlit IN TYPEOF (v1) THEN
      RETURN (ctmv(v1\complex_number_literal.imag_part));
    END_IF;
  ef_negate_c :
    IF cnlit IN TYPEOF (v1) THEN  parts(v1,p,q);  RETURN (makec(-p,-q));  END_IF;
  ef_reciprocal_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  t := p*p + q*q;  RETURN (makec(p/t,-q/t));
    END_IF;
  ef_add_c : BEGIN
    p := 0.0;  q := 0.0;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF cnlit IN TYPEOF (vlist[i]) THEN
        parts(vlist[i],r,s);  p := p + r;  q := q + s;
        REMOVE (vlist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (makec(p,q));  END_IF;
    IF p*p+q*q <> 0.0 THEN  INSERT (vlist, makec(p,q), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_subtract_c :
    IF (cnlit IN TYPEOF (v1)) AND (cnlit IN TYPEOF (v2)) THEN
      parts(v1,p,q);  parts(v2,r,s);  RETURN (makec(p-r,q-s));
    END_IF;
  ef_multiply_c : BEGIN
    p := 1.0;  q := 0.0;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF cnlit IN TYPEOF (vlist[i]) THEN
        parts(vlist[i],r,s);  p := p*r-q*s;  q := p*s+q*r;
        REMOVE (vlist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (makec(p,q));  END_IF;
    IF (p <> 1.0) OR (q <> 0.0) THEN  INSERT (vlist, makec(p,q), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_divide_c :
    IF (cnlit IN TYPEOF (v1)) AND (cnlit IN TYPEOF (v2)) THEN
      parts(v1,p,q);  parts(v2,r,s);  t := r*r+s*s;
      RETURN (makec((p*r+q*s)/t,(q*r-p*s)/t));
    END_IF;
  ef_exponentiate_c :
    IF (cnlit IN TYPEOF (v1)) AND (cnlit IN TYPEOF (v2)) THEN
      parts(v1,p,q);  parts(v2,r,s);  t := 0.5*LOG(p*p+q*q);  u := atan2(q,p);
      p := r*t-s*u;  q := r*u+s*t;  r := EXP(p);
      RETURN (makec(r*COS(q),r*SIN(q)));
    END_IF;
  ef_exponentiate_ci :
    IF (cnlit IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      parts(v1,p,q);  k := v2;  r := 1.0;  s := 0.0;
      REPEAT i := 1 TO ABS(k);  r := p*r-q*s;  s := p*s+q*r;  END_REPEAT;
```

© ISO 2002 — All rights reserved

```
      IF k < 0 THEN  t := r*r+s*s;  r := r/t;  s := -s/t;  END_IF;
      RETURN (makec(r,s));
    END_IF;
  ef_eq_c :
    IF (cnlit IN TYPEOF (v1)) AND (cnlit IN TYPEOF (v2)) THEN
      parts(v1,p,q);  parts(v2,r,s);  RETURN (ctmv((p = r) AND (q = s)));
    END_IF;
  ef_ne_c :
    IF (cnlit IN TYPEOF (v1)) AND (cnlit IN TYPEOF (v2)) THEN
      parts(v1,p,q);  parts(v2,r,s);  RETURN (ctmv((p <> r) OR (q <> s)));
    END_IF;
  ef_conjugate_c :
    IF cnlit IN TYPEOF (v1) THEN  parts(v1,p,q);  RETURN (makec(p,-q));  END_IF;
  ef_abs_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  RETURN (ctmv(SQRT(p*p+q*q)));
    END_IF;
  ef_arg_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  RETURN (ctmv(atan2(q,p)));
    END_IF;
  ef_cos_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  t := 0.5*EXP(-q);  u := 0.5*EXP(q);
      RETURN (makec((t+u)*COS(p),(t-u)*SIN(p)));
    END_IF;
  ef_exp_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  RETURN (makec(EXP(p)*COS(q),EXP(p)*SIN(q)));
    END_IF;
  ef_ln_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  RETURN (makec(0.5*LOG(p*p+q*q),atan2(q,p)));
    END_IF;
  ef_sin_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  t := 0.5*EXP(-q);  u := 0.5*EXP(q);
      RETURN (makec((t+u)*SIN(p),(u-t)*COS(p)));
    END_IF;
  ef_sqrt_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  t := SQRT(SQRT(p*p+q*q));  u := 0.5*atan2(q,p);
      RETURN (makec(t*COS(u),t*SIN(u)));
    END_IF;
  ef_tan_c :
    IF cnlit IN TYPEOF (v1) THEN
      parts(v1,p,q);  t := EXP(2.0*q) + EXP(-2.0*q) + 2.0*COS(2.0*p);
      RETURN (makec(2.0*SIN(2.0*p)/t,(EXP(-2.0*q)-EXP(2.0*q))/t));
    END_IF;
  -- ef_if_c : combined with ef_if
  ef_subscript_s :
    IF ('STRING' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      str := v1;  k := v2;  RETURN (ctmv(str[k]));
    END_IF;
  ef_eq_s :
    IF ('STRING' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
      str := v1;  st2 := v2;  RETURN (ctmv(str = st2));
    END_IF;
  ef_ne_s :
```


© ISO 2002 — All rights reserved

```
    IF ('STRING' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
      str := v1;  st2 := v2;  RETURN (ctmv(str <> st2));
    END_IF;
  ef_gt_s :
    IF ('STRING' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
      str := v1;  st2 := v2;  RETURN (ctmv(str > st2));
    END_IF;
  ef_lt_s :
    IF ('STRING' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
      str := v1;  st2 := v2;  RETURN (ctmv(str < st2));
    END_IF;
  ef_ge_s :
    IF ('STRING' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
      str := v1;  st2 := v2;  RETURN (ctmv(str >= st2));
    END_IF;
  ef_le_s :
    IF ('STRING' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
      str := v1;  st2 := v2;  RETURN (ctmv(str <= st2));
    END_IF;
  ef_subsequence_s :
    IF ('STRING' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) AND
      ('INTEGER' IN TYPEOF (v3)) THEN
      str := v1;  j := v2;  k := v3;  RETURN (ctmv(str[j:k]));
    END_IF;
  ef_concat_s : BEGIN
    str := '';
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'STRING' IN TYPEOF (vlist[i]) THEN
        st2 := vlist[i];  str := str + st2;
        REMOVE (vlist, i);
      ELSE IF str <> '' THEN
        INSERT (vlist, ctmv(str), i);
        str := '';
      END_IF;  END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(str));  END_IF;
    IF str <> '' THEN  INSERT (vlist, ctmv(str), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_size_s :
    IF 'STRING' IN TYPEOF (v1) THEN  str:=v1;  RETURN (ctmv(LENGTH(str)));  END_IF;
  ef_format :
    IF ('NUMBER' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
      RETURN (ctmv(FORMAT(v1,v2)));
    END_IF;
  ef_value :
    IF 'STRING' IN TYPEOF (v1) THEN  str:=v1;  RETURN (ctmv(VALUE(str)));  END_IF;
  ef_like :
    IF ('STRING' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
      RETURN (ctmv(v1 LIKE v2));
    END_IF;
  -- ef_if_s : combined with ef_if
  ef_subscript_b :
    IF ('BINARY' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      bin := v1;  k := v2;  RETURN (ctmv(bin[k]));
    END_IF;
  ef_eq_b :
    IF ('BINARY' IN TYPEOF (v1)) AND ('BINARY' IN TYPEOF (v2)) THEN
      bin := v1;  bi2 := v2;  RETURN (ctmv(bin = bi2));
```

© ISO 2002 — All rights reserved

```
    END_IF;
  ef_ne_b :
    IF ('BINARY' IN TYPEOF (v1)) AND ('BINARY' IN TYPEOF (v2)) THEN
      bin := v1;  bi2 := v2;  RETURN (ctmv(bin <> bi2));
    END_IF;
  ef_gt_b :
    IF ('BINARY' IN TYPEOF (v1)) AND ('BINARY' IN TYPEOF (v2)) THEN
      bin := v1;  bi2 := v2;  RETURN (ctmv(bin > bi2));
    END_IF;
  ef_lt_b :
    IF ('BINARY' IN TYPEOF (v1)) AND ('BINARY' IN TYPEOF (v2)) THEN
      bin := v1;  bi2 := v2;  RETURN (ctmv(bin < bi2));
    END_IF;
  ef_ge_b :
    IF ('BINARY' IN TYPEOF (v1)) AND ('BINARY' IN TYPEOF (v2)) THEN
      bin := v1;  bi2 := v2;  RETURN (ctmv(bin >= bi2));
    END_IF;
  ef_le_b :
    IF ('BINARY' IN TYPEOF (v1)) AND ('BINARY' IN TYPEOF (v2)) THEN
      bin := v1;  bi2 := v2;  RETURN (ctmv(bin <= bi2));
    END_IF;
  ef_subsequence_b :
    IF ('BINARY' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) AND
      ('INTEGER' IN TYPEOF (v3)) THEN
      bin := v1;  j := v2;  k := v3;  RETURN (ctmv(bin[j:k]));
    END_IF;
  ef_concat_b : BEGIN
    boo := FALSE;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'BINARY' IN TYPEOF (vlist[i]) THEN
        IF boo THEN  bi2 := vlist[i];  bin := bin + bi2;
        ELSE         bin := vlist[i];  boo := TRUE;  END_IF;
        REMOVE (vlist, i);
      ELSE IF boo THEN
        INSERT (vlist, ctmv(bin), i);
        boo := FALSE;
      END_IF;  END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(bin));  END_IF;
    IF boo THEN  INSERT (vlist, ctmv(bin), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_size_b :
    IF 'BINARY' IN TYPEOF (v1) THEN  bin:=v1;  RETURN (ctmv(BLENGTH(bin)));  END_IF;
  -- ef_if_b : combined with ef_if
  ef_subscript_t :
    IF ('LIST' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
      tpl := v1;  k := v2;  RETURN (ctmv(tpl[k]));
    END_IF;
  ef_eq_t :
    IF ('LIST' IN TYPEOF (v1)) AND ('LIST' IN TYPEOF (v2)) THEN
      lgc := equal_maths_values(v1,v2);
      IF lgc <> UNKNOWN THEN  RETURN (ctmv(lgc));  END_IF;
    END_IF;
  ef_ne_t :
    IF ('LIST' IN TYPEOF (v1)) AND ('LIST' IN TYPEOF (v2)) THEN
      lgc := equal_maths_values(v1,v2);
      IF lgc <> UNKNOWN THEN  RETURN (ctmv(NOT lgc));  END_IF;
    END_IF;
```

 © ISO 2002 — All rights reserved

```
ef_concat_t : BEGIN
  tpl := [];
  REPEAT i := SIZEOF (vlist) TO 1 BY -1;
    IF 'STRING' IN TYPEOF (vlist[i]) THEN
      tp2 := vlist[i];  tpl := tpl + tp2;
      REMOVE (vlist, i);
    ELSE IF SIZEOF (tpl) <> 0 THEN
      INSERT (vlist, ctmv(tpl), i);
      tpl := [];
    END_IF;  END_IF;
  END_REPEAT;
  IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(tpl));  END_IF;
  IF SIZEOF (tpl) <> 0 THEN  INSERT (vlist, ctmv(tpl), 0);  END_IF;
  IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
  END;
ef_size_t :
  IF 'LIST' IN TYPEOF (v1) THEN  tpl:=v1;  RETURN (ctmv(SIZEOF(tpl)));  END_IF;
ef_entuple :
  RETURN (ctmv(vlist));
ef_detuple :  -- This can have multiple outputs, but the expression only
              -- denotes the first.
  IF 'LIST' IN TYPEOF (v1) THEN  tpl:=v1;  RETURN (ctmv(tpl[1]));  END_IF;
ef_insert :
  IF ('LIST' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v3)) THEN
    tpl := v1;  k := v3;  INSERT (tpl, v2, k);  RETURN (ctmv(tpl));
  END_IF;
ef_remove :
  IF ('LIST' IN TYPEOF (v1)) AND ('INTEGER' IN TYPEOF (v2)) THEN
    tpl := v1;  k := v2;  REMOVE (tpl, k);  RETURN (ctmv(tpl));
  END_IF;
-- ef_if_t : combined with ef_if
ef_sum_it :
  IF good_t(v1,'INTEGER') THEN
    tpl := v1;  j := 0;
    REPEAT i := 1 TO SIZEOF (tpl);  j := j + tpl[i];  END_REPEAT;
    RETURN (ctmv(j));
  END_IF;
ef_product_it :
  IF good_t(v1,'INTEGER') THEN
    tpl := v1;  j := 1;
    REPEAT i := 1 TO SIZEOF (tpl);  j := j * tpl[i];  END_REPEAT;
    RETURN (ctmv(j));
  END_IF;
ef_add_it : BEGIN
  boo := FALSE;
  REPEAT i := SIZEOF (vlist) TO 1 BY -1;
    IF good_t(vlist[i],'INTEGER') THEN
      IF NOT boo THEN  tpl := vlist[i];  boo := TRUE;
      ELSE
        tp2 := vlist[i];
        IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
        REPEAT l := 1 TO SIZEOF (tpl);  tpl[j] := tpl[j] + tp2[j];  END_REPEAT;
      END_IF;
      REMOVE (vlist, i);
    END_IF;
  END_REPEAT;
  IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(tpl));  END_IF;
  IF boo THEN  INSERT (vlist, ctmv(tpl), 0);  END_IF;
  IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
```

© ISO 2002 — All rights reserved

```
    END;
  ef_subtract_it :
    IF good_t(v1,'INTEGER') AND good_t(v2,'INTEGER') THEN
      tpl := v1;  tp2 := v2;
      IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
      REPEAT i := 1 TO SIZEOF (tpl);  tpl[i] := tpl[i] - tp2[i];  END_REPEAT;
      RETURN (ctmv(tpl));
    END_IF;
  ef_scalar_mult_it :
    IF ('INTEGER' IN TYPEOF (v1)) AND good_t(v2,'INTEGER') THEN
      j := v1;  tpl := v2;
      REPEAT i := 1 TO SIZEOF (tpl);  tpl[i] := j * tpl[i];  END_REPEAT;
      RETURN (ctmv(tpl));
    END_IF;
  ef_dot_prod_it :
    IF good_t(v1,'INTEGER') AND good_t(v2,'INTEGER') THEN
      tpl := v1;  tp2 := v2;  j := 0;
      IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
      REPEAT i := 1 TO SIZEOF (tpl);  j := j + tpl[i] * tp2[i];  END_REPEAT;
      RETURN (ctmv(j));
    END_IF;
  ef_sum_rt :
    IF good_t(v1,'REAL') THEN
      tpl := v1;  r := 0.0;
      REPEAT i := 1 TO SIZEOF (tpl);  r := r + tpl[i];  END_REPEAT;
      RETURN (ctmv(r));
    END_IF;
  ef_product_rt :
    IF good_t(v1,'REAL') THEN
      tpl := v1;  r := 1.0;
      REPEAT i := 1 TO SIZEOF (tpl);  r := r * tpl[i];  END_REPEAT;
      RETURN (ctmv(r));
    END_IF;
  ef_add_rt : BEGIN
    boo := FALSE;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF good_t(vlist[i],'REAL') THEN
        IF NOT boo THEN  tpl := vlist[i];  boo := TRUE;
        ELSE
          tp2 := vlist[i];
          IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
          REPEAT l := 1 TO SIZEOF (tpl);  tpl[j] := tpl[j] + tp2[j];  END_REPEAT;
        END_IF;
        REMOVE (vlist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(tpl));  END_IF;
    IF boo THEN  INSERT (vlist, ctmv(tpl), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_subtract_rt :
    IF good_t(v1,'REAL') AND good_t(v2,'REAL') THEN
      tpl := v1;  tp2 := v2;
      IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
      REPEAT i := 1 TO SIZEOF (tpl);  tpl[i] := tpl[i] - tp2[i];  END_REPEAT;
      RETURN (ctmv(tpl));
    END_IF;
  ef_scalar_mult_rt :
    IF ('REAL' IN TYPEOF (v1)) AND good_t(v2,'REAL') THEN
```

 © ISO 2002 — All rights reserved

```
      r := v1;  tpl := v2;
      REPEAT i := 1 TO SIZEOF (tpl);  tpl[i] := r * tpl[i];  END_REPEAT;
      RETURN (ctmv(tpl));
    END_IF;
  ef_dot_prod_rt :
    IF good_t(v1,'REAL') AND good_t(v2,'REAL') THEN
      tpl := v1;  tp2 := v2;  r := 0;
      IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
      REPEAT i := 1 TO SIZEOF (tpl);  r := r + tpl[i] * tp2[i];  END_REPEAT;
      RETURN (ctmv(r));
    END_IF;
  ef_norm_rt :
    IF good_t(v1,'REAL') THEN
      tpl := v1;  r := 0.0;
      REPEAT i := 1 TO SIZEOF (tpl);  r := r + tpl[i]*tpl[i];  END_REPEAT;
      RETURN (ctmv(SQRT(r)));
    END_IF;
  ef_sum_ct :
    IF good_t(v1,cnlit) THEN
      tpl := v1;  p := 0.0;  q := 0.0;
      REPEAT i:=1 TO SIZEOF (tpl);  parts(tpl[i],r,s);  p:=p+r;  q:=q+s;  END_REPEAT;
      RETURN (makec(p,q));
    END_IF;
  ef_product_ct :
    IF good_t(v1,cnlit) THEN
      tpl := v1;  p := 1.0;  q := 0.0;
      REPEAT i := 1 TO SIZEOF (tpl);
        parts(tpl[i],r,s);  p := p*r-q*s;  q := p*s+q*r;
      END_REPEAT;
      RETURN (makec(p,q));
    END_IF;
  ef_add_ct : BEGIN
    boo := FALSE;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF good_t(vlist[i],cnlit) THEN
        IF NOT boo THEN  tpl := vlist[i];  boo := TRUE;
        ELSE
          tp2 := vlist[i];
          IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
          REPEAT l := 1 TO SIZEOF (tpl);
            parts(tpl[j],p,q); parts(tp2[j],r,s);  tpl[j] := makec(p+r,q+s);
          END_REPEAT;
        END_IF;
        REMOVE (vlist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (vlist) = 0 THEN  RETURN (ctmv(tpl));  END_IF;
    IF boo THEN  INSERT (vlist, ctmv(tpl), 0);  END_IF;
    IF SIZEOF (vlist) = 1 THEN  RETURN (vlist[1]);  END_IF;
    END;
  ef_subtract_ct :
    IF good_t(v1,cnlit) AND good_t(v2,cnlit) THEN
      tpl := v1;  tp2 := v2;
      IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
      REPEAT i := 1 TO SIZEOF (tpl);
        parts(tpl[i],p,q);  parts(tp2[i],r,s);  tpl[i] := makec(p-r,q-s);
      END_REPEAT;
      RETURN (ctmv(tpl));
    END_IF;
```

© ISO 2002 — All rights reserved

```
    ef_scalar_mult_ct :
      IF (cnlit IN TYPEOF (v1)) AND good_t(v2,cnlit) THEN
        parts(v1,p,q);  tpl := v2;
        REPEAT i := 1 TO SIZEOF (tpl);
          parts(tpl[i],r,s);  tpl[i] := makec(p*r-q*s,p*s+q*r);
        END_REPEAT;
        RETURN (ctmv(tpl));
      END_IF;
    ef_dot_prod_ct :
      IF good_t(v1,cnlit) AND good_t(v2,cnlit) THEN
        tpl := v1;  tp2 := v2;  t := 0.0;  u := 0.0;
        IF SIZEOF (tpl) <> SIZEOF (tp2) THEN  RETURN (?);  END_IF;
        REPEAT i := 1 TO SIZEOF (tpl);
          parts(tpl[i],p,q);  parts(tp2[i],r,s);  t := t + p*r+q*s;  u := u + q*r-p*s;
        END_REPEAT;
        RETURN (makec(t,u));
      END_IF;
    ef_norm_ct :
      IF good_t(v1,cnlit) THEN
        tpl := v1;  r := 0.0;
        REPEAT i := 1 TO SIZEOF (tpl);  parts(tpl[i],p,q);  r:=r+p*p+q*q;  END_REPEAT;
        RETURN (ctmv(SQRT(r)));
      END_IF;
    ef_if, ef_if_i, ef_if_r, ef_if_c, ef_if_s, ef_if_b, ef_if_t :
      IF 'LOGICAL' IN TYPEOF (v1) THEN
        lgc := v1;  IF lgc THEN  RETURN (v2);  ELSE  RETURN (v3);  END_IF;
      END_IF;
    ef_ensemble :   -- (mem + vlist) effectively converts list to set
      RETURN (make_finite_space(mem + vlist));
    ef_member_of :
      IF (schema_prefix + 'MATHS_SPACE') IN TYPEOF (v2) THEN
        lgc := member_of(v1,v2);
        IF lgc <> UNKNOWN THEN  RETURN (ctmv(lgc));  END_IF;
      END_IF;
    END_CASE;
    RETURN (make_function_application(expr.func,vlist));
  END_IF;
  IF 'ABSTRACTED_EXPRESSION_FUNCTION' IN types THEN
    gexpr := substitute(expr.func\abstracted_expression_function.expr,
      expr.func\quantifier_expression.variables,vlist);
    RETURN (simplify_generic_expression(gexpr));
  END_IF;
  IF 'FINITE_FUNCTION' IN types THEN
    pairs := expr.func\finite_function.pairs;
    REPEAT i := 1 TO SIZEOF (pairs);
      IF equal_maths_values(vlist[1],pairs[i][1]) THEN
        RETURN (simplify_maths_value(pairs[i][2]));
      END_IF;
    END_REPEAT;
    RETURN (make_function_application(expr.func,vlist));
  END_IF;
  RETURN (expr);
END_FUNCTION;  -- simplify_function_application
(*
```

Argument definitions:

expr: (input) The function application expression to be simplified.

© ISO 2002 — All rights reserved

return: (output) A **maths_value** denoting the same mathematical object as the input expression.

4.6.137 simplify_generic_expression

This function attempts to simplify a **generic_expression** to obtain a simpler **maths_value**. When no simpler expression can be identified, the input expression is returned unchanged.

EXPRESS specification:

```
*)
FUNCTION simplify_generic_expression(expr : generic_expression) : maths_value;
  FUNCTION restore_unary(expr : unary_generic_expression;
                         opnd : generic_expression) : generic_expression;
    expr.operand := opnd;
    RETURN (expr);
  END_FUNCTION;  -- restore_unary
  FUNCTION restore_binary(expr       : binary_generic_expression;
                          opd1, opd2 : generic_expression) : generic_expression;
    expr.operands[1] := opd1;
    expr.operands[2] := opd2;
    RETURN (expr);
  END_FUNCTION;  -- restore_binary
  FUNCTION restore_mulary(expr : multiple_arity_generic_expression;
                          ops  : LIST OF generic_expression) : generic_expression;
    expr.operands := ops;
    RETURN (expr);
  END_FUNCTION;  -- restore_mulary
  FUNCTION make_number_literal(nmb : NUMBER) : generic_literal;
    IF 'INTEGER' IN TYPEOF (nmb) THEN  RETURN (make_int_literal(nmb));  END_IF;
    RETURN (make_real_literal(nmb));
  END_FUNCTION;  -- make_number_literal;
  LOCAL
    types : SET OF STRING := stripped_typeof (expr);
    v1, v2 : maths_value;
    vlist : LIST OF maths_value := [];
    op1, op2 : generic_expression;
    oplist : LIST OF generic_expression := [];
    opnds : LIST [2:?] OF generic_expression;
    n, m : INTEGER;
    finfun : maths_function_select;
    boo : BOOLEAN;
    str : STRING;
    nmb : NUMBER;
  END_LOCAL;
  -- Unwrap the elementary kinds of literals
  IF 'INT_LITERAL' IN types THEN
    RETURN (convert_to_maths_value (expr\int_literal.the_value));
  END_IF;
  IF 'REAL_LITERAL' IN types THEN
    RETURN (convert_to_maths_value (expr\real_literal.the_value));
  END_IF;
  IF 'BOOLEAN_LITERAL' IN types THEN
    RETURN (convert_to_maths_value (expr\boolean_literal.the_value));
  END_IF;
  IF 'STRING_LITERAL' IN types THEN
```

© ISO 2002 — All rights reserved

```
    RETURN (convert_to_maths_value (expr\string_literal.the_value));
  END_IF;
  IF 'COMPLEX_NUMBER_LITERAL' IN types THEN
    RETURN (expr);  -- No simpler expression available
  END_IF;
  IF 'LOGICAL_LITERAL' IN types THEN
    RETURN (convert_to_maths_value (expr\logical_literal.lit_value));
  END_IF;
  IF 'BINARY_LITERAL' IN types THEN
    RETURN (convert_to_maths_value (expr\binary_literal.lit_value));
  END_IF;
  IF 'MATHS_ENUM_LITERAL' IN types THEN
    RETURN (expr\maths_enum_literal.lit_value);
  END_IF;
  IF 'REAL_TUPLE_LITERAL' IN types THEN
    RETURN (convert_to_maths_value (expr\real_tuple_literal.lit_value));
  END_IF;
  IF 'INTEGER_TUPLE_LITERAL' IN types THEN
    RETURN (convert_to_maths_value (expr\integer_tuple_literal.lit_value));
  END_IF;
  IF 'ATOM_BASED_LITERAL' IN types THEN
    RETURN (expr\atom_based_literal.lit_value);
  END_IF;
  IF 'MATHS_TUPLE_LITERAL' IN types THEN
    RETURN (convert_to_maths_value (expr\maths_tuple_literal.lit_value));
  END_IF;
  -- Simplify one special class of literals
  IF 'MATHS_SPACE' IN types THEN
    RETURN (simplify_maths_space(expr));
  END_IF;
  -- Simplify one special kind of expression
  IF 'FUNCTION_APPLICATION' IN types THEN
    RETURN (simplify_function_application(expr));
  END_IF;
  -- Separate and simplify the operands
  IF 'UNARY_GENERIC_EXPRESSION' IN types THEN
    v1 := simplify_generic_expression(expr\unary_generic_expression.operand);
    op1 := convert_to_operand(v1);
  END_IF;
  IF 'BINARY_GENERIC_EXPRESSION' IN types THEN
    v1 := simplify_generic_expression(expr\binary_generic_expression.operands[1]);
    op1 := convert_to_operand(v1);
    v2 := simplify_generic_expression(expr\binary_generic_expression.operands[2]);
    op2 := convert_to_operand(v2);
  END_IF;
  IF 'MULTIPLE_ARITY_GENERIC_EXPRESSION' IN types THEN
    opnds := expr\multiple_arity_generic_expression.operands;
    REPEAT i := 1 TO SIZEOF (opnds);
      v1 := simplify_generic_expression(opnds[i]);
      INSERT (vlist, v1, i-1);
      INSERT (oplist, convert_to_operand(v1), i-1);
    END_REPEAT;
  END_IF;
  -- Simplify the one kind of maths_function which derives its operands.
  IF 'PARALLEL_COMPOSED_FUNCTION' IN types THEN
    v1 := vlist[1];
    n := SIZEOF (vlist);
    finfun := vlist[n];
    REMOVE (vlist, n);
```

 © ISO 2002 — All rights reserved

```
    REMOVE (vlist, 1);
    RETURN (make_parallel_composed_function(v1,vlist,finfun));
  END_IF;
  -- Simplify individual kinds of expressions.  It is not necessary to cover all cases.
  IF ('ABS_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (ABS(v1)));
  END_IF;
  IF ('ACOS_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (ACOS(v1)));
  END_IF;
  IF 'AND_EXPRESSION' IN types THEN
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'BOOLEAN' IN TYPEOF (vlist[i]) THEN
        boo := vlist[i];
        IF NOT boo THEN  RETURN (convert_to_maths_value(FALSE));  END_IF;
        REMOVE (oplist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (oplist) = 0 THEN  RETURN (convert_to_maths_value(TRUE));  END_IF;
    IF SIZEOF (oplist) = 1 THEN  RETURN (oplist[1]);  END_IF;
  END_IF;
  IF ('ASIN_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (ASIN(v1)));
  END_IF;
  IF ('ATAN_EXPRESSION' IN types) AND
    ('NUMBER' IN TYPEOF (v1)) AND ('NUMBER' IN TYPEOF (v2)) THEN
    RETURN (convert_to_maths_value (ATAN(v1,v2)));
  END_IF;
  IF ('COMPARISON_EXPRESSION' IN types) AND (
    (('NUMBER' IN TYPEOF (v1)) AND ('NUMBER' IN TYPEOF (v2))) OR
    (('STRING' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2))) OR
    (('BOOLEAN' IN TYPEOF (v1)) AND ('BOOLEAN' IN TYPEOF (v2))) ) THEN
    IF      'COMPARISON_EQUAL'         IN types THEN  boo := bool(v1 = v2);
    ELSE IF 'COMPARISON_GREATER'       IN types THEN  boo := bool(v1 > v2);
    ELSE IF 'COMPARISON_GREATER_EQUAL' IN types THEN  boo := bool(v1 >= v2);
    ELSE IF 'COMPARISON_LESS'          IN types THEN  boo := bool(v1 < v2);
    ELSE IF 'COMPARISON_LESS_EQUAL'    IN types THEN  boo := bool(v1 <= v2);
    ELSE IF 'COMPARISON_NOT_EQUAL'     IN types THEN  boo := bool(v1 <> v2);
    ELSE IF 'LIKE_EXPRESSION'          IN types THEN  boo := bool(v1 LIKE v2);
    ELSE  RETURN (?);  -- Unreachable
    END_IF;  END_IF;  END_IF;  END_IF;  END_IF;  END_IF;  END_IF;
    RETURN (convert_to_maths_value (boo));
  END_IF;
  IF 'CONCAT_EXPRESSION' IN types THEN
    str := '';
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'STRING' IN TYPEOF (vlist[i]) THEN
        str := vlist[i] + str;
        REMOVE (oplist, i);
      ELSE IF LENGTH(str) > 0 THEN
        INSERT (oplist, make_string_literal(str), i);
        str := '';
      END_IF;  END_IF;
    END_REPEAT;
    IF SIZEOF (oplist) = 0 THEN  RETURN (convert_to_maths_value(str));  END_IF;
    IF LENGTH(str) > 0 THEN  INSERT (oplist, make_string_literal(str), 0);  END_IF;
    IF SIZEOF (oplist) = 1 THEN  RETURN (oplist[1]);  END_IF;
  END_IF;
  IF ('COS_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
```

© ISO 2002 — All rights reserved

```
    RETURN (convert_to_maths_value (COS(v1)));
  END_IF;
  IF ('DIV_EXPRESSION' IN types) AND
    ('NUMBER' IN TYPEOF (v1)) AND ('NUMBER' IN TYPEOF (v2)) THEN
    RETURN (convert_to_maths_value (v1 DIV v2));
  END_IF;
  IF 'EQUALS_EXPRESSION' IN types THEN
    opnds := expr\binary_generic_expression.operands;
    RETURN (convert_to_maths_value (opnds[1] :=: opnds[2]));
  END_IF;
  IF ('EXP_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (EXP(v1)));
  END_IF;
  IF ('FORMAT_EXPRESSION' IN types) AND
    ('NUMBER' IN TYPEOF (v1)) AND ('STRING' IN TYPEOF (v2)) THEN
    RETURN (convert_to_maths_value (FORMAT(v1,v2)));
  END_IF;
  IF ('INDEX_EXPRESSION' IN types) AND
    ('STRING' IN TYPEOF (v1)) AND ('NUMBER' IN TYPEOF (v2)) THEN
    str := v1;  n := v2;
    RETURN (convert_to_maths_value (str[n]));
  END_IF;
  IF ('INT_VALUE_EXPRESSION' IN types) AND ('STRING' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (VALUE(v1)));
  END_IF;
  IF 'INTERVAL_EXPRESSION' IN types THEN
    str := '';
    IF 'NUMBER'  IN TYPEOF (vlist[1]) THEN str := 'NUMBER';   END_IF;
    IF 'STRING'  IN TYPEOF (vlist[1]) THEN str := 'STRING';   END_IF;
    IF 'BOOLEAN' IN TYPEOF (vlist[1]) THEN str := 'BOOLEAN';  END_IF;
    IF (LENGTH (str) > 0) AND (str IN TYPEOF (vlist[2])) AND
      (str IN TYPEOF (vlist[3])) THEN
      RETURN (convert_to_maths_value ({vlist[1] <= vlist[2] <= vlist[3]}));
    END_IF;
  END_IF;
  IF ('LENGTH_EXPRESSION' IN types) AND ('STRING' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (LENGTH(v1)));
  END_IF;
  IF ('LOG_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (LOG(v1)));
  END_IF;
  IF ('LOG10_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (LOG10(v1)));
  END_IF;
  IF ('LOG2_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (LOG2(v1)));
  END_IF;
  IF 'MAXIMUM_EXPRESSION' IN types THEN
    boo := FALSE;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'NUMBER' IN TYPEOF (vlist[i]) THEN
        IF boo THEN
          IF nmb < vlist[i] THEN  nmb := vlist[i];  END_IF;
        ELSE
          nmb := vlist[i];  boo := TRUE;
        END_IF;
        REMOVE (oplist, i);
      END_IF;
    END_REPEAT;
```

 © ISO 2002 — All rights reserved

```
    IF SIZEOF (oplist) = 0 THEN  RETURN (convert_to_maths_value(nmb));  END_IF;
    IF boo THEN  INSERT (oplist, make_number_literal(nmb), 0);  END_IF;
  END_IF;
  IF 'MINIMUM_EXPRESSION' IN types THEN
    boo := FALSE;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'NUMBER' IN TYPEOF (vlist[i]) THEN
        IF boo THEN
          IF nmb > vlist[i] THEN  nmb := vlist[i];  END_IF;
        ELSE
          nmb := vlist[i];  boo := TRUE;
        END_IF;
        REMOVE (oplist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (oplist) = 0 THEN  RETURN (convert_to_maths_value(nmb));  END_IF;
    IF boo THEN  INSERT (oplist, make_number_literal(nmb), 0);  END_IF;
  END_IF;
  IF ('MINUS_EXPRESSION' IN types) AND
    ('NUMBER' IN TYPEOF (v1)) AND ('NUMBER' IN TYPEOF (v2)) THEN
    RETURN (convert_to_maths_value (v1 - v2));
  END_IF;
  IF ('MOD_EXPRESSION' IN types) AND
    ('NUMBER' IN TYPEOF (v1)) AND ('NUMBER' IN TYPEOF (v2)) THEN
    RETURN (convert_to_maths_value (v1 MOD v2));
  END_IF;
  IF 'MULT_EXPRESSION' IN types THEN
    nmb := 1;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'NUMBER' IN TYPEOF (vlist[i]) THEN
        nmb := nmb * vlist[i];
        REMOVE (oplist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (oplist) = 0 THEN  RETURN (convert_to_maths_value(nmb));  END_IF;
    IF nmb <> 1 THEN  INSERT (oplist, make_number_literal(nmb), 0);  END_IF;
    IF SIZEOF (oplist) = 1 THEN  RETURN (oplist[1]);  END_IF;
  END_IF;
  IF ('NOT_EXPRESSION' IN types) AND ('BOOLEAN' IN TYPEOF (v1)) THEN
    boo := v1;
    RETURN (convert_to_maths_value (NOT(boo)));
  END_IF;
  IF ('ODD_EXPRESSION' IN types) AND ('INTEGER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (ODD(v1)));
  END_IF;
  IF 'OR_EXPRESSION' IN types THEN
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
      IF 'BOOLEAN' IN TYPEOF (vlist[i]) THEN
        boo := vlist[i];
        IF boo THEN  RETURN (convert_to_maths_value(TRUE));  END_IF;
        REMOVE (oplist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (oplist) = 0 THEN  RETURN (convert_to_maths_value(FALSE));  END_IF;
    IF SIZEOF (oplist) = 1 THEN  RETURN (oplist[1]);  END_IF;
  END_IF;
  IF 'PLUS_EXPRESSION' IN types THEN
    nmb := 0;
    REPEAT i := SIZEOF (vlist) TO 1 BY -1;
```

© ISO 2002 — All rights reserved

```
      IF 'NUMBER' IN TYPEOF (vlist[i]) THEN
        nmb := nmb + vlist[i];
        REMOVE (oplist, i);
      END_IF;
    END_REPEAT;
    IF SIZEOF (oplist) = 0 THEN  RETURN (convert_to_maths_value(nmb));  END_IF;
    IF nmb <> 0 THEN  INSERT (oplist, make_number_literal(nmb), 0);  END_IF;
    IF SIZEOF (oplist) = 1 THEN  RETURN (oplist[1]);  END_IF;
  END_IF;
  IF ('POWER_EXPRESSION' IN types) AND
    ('NUMBER' IN TYPEOF (v1)) AND ('NUMBER' IN TYPEOF (v2)) THEN
    RETURN (convert_to_maths_value (v1 ** v2));
  END_IF;
  IF ('SIN_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (SIN(v1)));
  END_IF;
  IF ('SLASH_EXPRESSION' IN types) AND
    ('NUMBER' IN TYPEOF (v1)) AND ('NUMBER' IN TYPEOF (v2)) THEN
    RETURN (convert_to_maths_value (v1 / v2));
  END_IF;
  IF ('SQUARE_ROOT_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (SQRT(v1)));
  END_IF;
  IF ('SUBSTRING_EXPRESSION' IN types) AND
    ('STRING' IN TYPEOF (vlist[1])) AND ('NUMBER' IN TYPEOF (vlist[2])) AND
    ('NUMBER' IN TYPEOF (vlist[3])) THEN
    str := vlist[1];  n := vlist[2];  m := vlist[3];
    RETURN (convert_to_maths_value (str[n:m]));
  END_IF;
  IF ('TAN_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (TAN(v1)));
  END_IF;
  IF ('UNARY_MINUS_EXPRESSION' IN types) AND ('NUMBER' IN TYPEOF (v1)) THEN
    nmb := v1;
    RETURN (convert_to_maths_value (-nmb));
  END_IF;
  IF ('VALUE_EXPRESSION' IN types) AND ('STRING' IN TYPEOF (v1)) THEN
    RETURN (convert_to_maths_value (VALUE(v1)));
  END_IF;
  IF ('XOR_EXPRESSION' IN types) AND
    ('BOOLEAN' IN TYPEOF (v1)) AND ('BOOLEAN' IN TYPEOF (v2)) THEN
    RETURN (convert_to_maths_value (v1 XOR v2));
  END_IF;
  -- No special simplification defined, return same with simplified operands.
  IF 'UNARY_GENERIC_EXPRESSION' IN types THEN
    RETURN (restore_unary(expr,op1));
  END_IF;
  IF 'BINARY_GENERIC_EXPRESSION' IN types THEN
    RETURN (restore_binary(expr,op1,op2));
  END_IF;
  IF 'MULTIPLE_ARITY_GENERIC_EXPRESSION' IN types THEN
    RETURN (restore_mulary(expr,oplist));
  END_IF;
  -- Should be unreachable, but for safety, return unsimplified expression.
  RETURN (expr);
END_FUNCTION;  -- simplify_generic_expression
(*
```

 © ISO 2002 — All rights reserved

Argument definitions:

expr: (input) The expression to be simplified.

return: (output) A **maths_value** denoting the same mathematical object as the input expression.

4.6.138 simplify_maths_space

This function attempts to simplify a **maths_space** to obtain a simpler instance of **maths_space**. The only cases where any simplification is possible are those involving finite spaces whose members are given by expressions which can be simplified. In any case, an instance of **maths_space** is returned which denotes the same mathematical space as that denoted by the input.

EXPRESS specification:

```
*)
FUNCTION simplify_maths_space(spc : maths_space) : maths_space;
  LOCAL
    stypes : SET OF STRING := stripped_typeof (spc);
    sset : SET OF maths_value;
    zset : SET OF maths_value := [];
    zval : maths_value;
    zspc : maths_space;
    zallint : BOOLEAN := TRUE;
    zint, zmin, zmax : INTEGER;
    factors : LIST OF maths_space;
    zfactors : LIST OF maths_space := [];
    rspc : maths_space;
  END_LOCAL;
  IF 'FINITE_SPACE' IN stypes THEN
    sset := spc\finite_space.members;
    REPEAT i := 1 TO SIZEOF (sset);
      zval := simplify_maths_value(sset[i]);
      zset := zset + [zval];
      IF zallint AND ('INTEGER' IN TYPEOF (zval)) THEN
        zint := zval;
        IF i = 1 THEN
          zmin := zint;
          zmax := zint;
        ELSE
          IF zint < zmin THEN
            zmin := zint;
          END_IF;
          IF zint > zmax THEN
            zmax := zint;
          END_IF;
        END_IF;
      ELSE
        zallint := FALSE;
      END_IF;
    END_REPEAT;
    IF zallint AND (SIZEOF(zset) = zmax-zmin+1) THEN
      RETURN (make_finite_integer_interval(zmin,zmax));
    END_IF;
```

© ISO 2002 — All rights reserved

```
    RETURN (make_finite_space(zset));
  END_IF;
  IF 'UNIFORM_PRODUCT_SPACE' IN stypes THEN
    zspc := simplify_maths_space(spc\uniform_product_space.base);
    RETURN (make_uniform_product_space(zspc,spc\uniform_product_space.exponent));
  END_IF;
  IF 'LISTED_PRODUCT_SPACE' IN stypes THEN
    factors := spc\listed_product_space.factors;
    REPEAT i := 1 TO SIZEOF (factors);
      INSERT (zfactors, simplify_maths_space(factors[i]), i-1);
    END_REPEAT;
    RETURN (make_listed_product_space(zfactors));
  END_IF;
  IF 'EXTENDED_TUPLE_SPACE' IN stypes THEN
    zspc := simplify_maths_space(spc\extended_tuple_space.base);
    rspc := simplify_maths_space(spc\extended_tuple_space.extender);
    RETURN (make_extended_tuple_space(zspc,rspc));
  END_IF;
  IF 'FUNCTION_SPACE' IN stypes THEN
    zspc := simplify_maths_space(spc\function_space.domain_argument);
    rspc := simplify_maths_space(spc\function_space.range_argument);
    RETURN (make_function_space(spc\function_space.domain_constraint,zspc,
      spc\function_space.range_constraint,rspc));
  END_IF;
  RETURN (spc);
END_FUNCTION;  -- simplify_maths_space
(*
```

Argument definitions:

spc: (input) The space to be simplified.

return: (output) A **maths_space** denoting the same mathematical object as the input space, but possibly with a simpler structure.

4.6.139 simplify_maths_value

This function attempts to simplify a **maths_value** to obtain a simpler instance of **maths_value**. The only cases where simplification is possible are those involving generic expressions. In any case, a **maths_value** is returned which denotes the same mathematical value as that denoted by the input.

EXPRESS specification:

```
*)
FUNCTION simplify_maths_value(val : maths_value) : maths_value;
  LOCAL
    vtypes : SET OF STRING := stripped_typeof(val);
    vlist : LIST OF maths_value;
    nlist : LIST OF maths_value := [];
  END_LOCAL;
  IF 'GENERIC_EXPRESSION' IN vtypes THEN
    RETURN (simplify_generic_expression(val));
  END_IF;
  IF 'LIST' IN vtypes THEN
```


© ISO 2002 — All rights reserved

```
    vlist := val;
    REPEAT i := 1 TO SIZEOF (vlist);
      INSERT (nlist, simplify_maths_value(vlist[i]), i-1);
    END_REPEAT;
    RETURN (convert_to_maths_value(nlist));
  END_IF;
  RETURN (val);
END_FUNCTION;  -- simplify_maths_value
(*
```

Argument definitions:

val: (input) The **maths_value** to be simplified.

return: (output) A **maths_value** denoting the same mathematical object as the input value, but possibly with a simpler structure.

4.6.140 singleton_member_of

This function returns the single member of a mathematical space which has only one member. Otherwise, it returns the indeterminate value.

EXPRESS specification:

```
*)
FUNCTION singleton_member_of(spc : maths_space) : maths_value;
  LOCAL
    types : SET OF STRING := stripped_typeof (spc);
  END_LOCAL;
  IF 'FINITE_SPACE' IN types THEN
    IF SIZEOF (spc\finite_space.members) = 1 THEN
      RETURN (spc\finite_space.members[1]);
    END_IF;
    RETURN (?);
  END_IF;
  IF 'FINITE_INTEGER_INTERVAL' IN types THEN
    IF spc\finite_integer_interval.size = 1 THEN
      RETURN (spc\finite_integer_interval.min);
    END_IF;
    RETURN (?);
  END_IF;
  RETURN (?);
END_FUNCTION;  -- singleton_member_of
(*
```

Argument definitions:

spc: (input) The space whose sole element is to be returned.

return: (output) The sole member of **spc**, or else the indeterminate value.

© ISO 2002 — All rights reserved

4.6.141 space_dimension

This function returns the dimension of a tuple space. In the case of a **product_space**, the dimension is the number of factors in the cartesian product, which is equivalent to the length of its members. In the case of an **extended_tuple_space**, the dimension is the minimum length of any of its members.

NOTE An **extended_tuple_space** has no upper limit on the length of its members.

EXPRESS specification:

```
*)
FUNCTION space_dimension(tspace : tuple_space) : nonnegative_integer;
  LOCAL
    types : SET OF STRING := TYPEOF (tspace);
  END_LOCAL;
  IF (schema_prefix + 'UNIFORM_PRODUCT_SPACE') IN types THEN
    RETURN (tspace\uniform_product_space.exponent);
  END_IF;
  IF (schema_prefix + 'LISTED_PRODUCT_SPACE') IN types THEN
    RETURN (SIZEOF (tspace\listed_product_space.factors));
  END_IF;
  IF (schema_prefix + 'EXTENDED_TUPLE_SPACE') IN types THEN
    -- In the case of an extended_tuple_space, the minimum dimension is returned.
    RETURN (space_dimension (tspace\extended_tuple_space.base));
  END_IF;
  -- Should be unreachable
  RETURN (?);
END_FUNCTION;  -- space_dimension
(*
```

Argument definitions:

tspace: (input) The tuple space whose dimension is to be determined.

return: (output) The dimension of the tuple space.

4.6.142 space_is_continuum

This function determines whether or not a space is a continuum, that is, whether or not the space is a subspace of the reals, a subspace of the complex numbers, or a cartesian product of spaces which are continua.

EXPRESS specification:

```
*)
FUNCTION space_is_continuum(space : maths_space) : BOOLEAN;
  LOCAL
    typenames : SET OF STRING := TYPEOF (space);
    factors : LIST OF maths_space;
  END_LOCAL;
  IF NOT EXISTS (space) THEN
```

 © ISO 2002 — All rights reserved

```
    RETURN (FALSE);
  END_IF;
  IF subspace_of_es(space,es_reals) OR subspace_of_es(space,es_complex_numbers) THEN
    RETURN (TRUE);
  END_IF;
  IF (schema_prefix + 'UNIFORM_PRODUCT_SPACE') IN typenames THEN
    RETURN (space_is_continuum(space\uniform_product_space.base));
  END_IF;
  IF (schema_prefix + 'LISTED_PRODUCT_SPACE') IN typenames THEN
    factors := space\listed_product_space.factors;
    IF SIZEOF(factors) = 0 THEN
      RETURN (FALSE);
    END_IF;
    REPEAT i := 1 TO SIZEOF (factors);
      IF NOT space_is_continuum(factors[i]) THEN
        RETURN (FALSE);
      END_IF;
    END_REPEAT;
    RETURN (TRUE);
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- space_is_continuum
(*
```

Argument definitions:

space: (input) The space to be tested for being a continuum.

return: (output) A BOOLEAN value indicating whether or not the space is a continuum.

4.6.143 space_is_singleton

This function determines whether or not its argument represents a mathematical space with a single member or not.

EXPRESS specification:

```
*)
FUNCTION space_is_singleton(spc : maths_space) : BOOLEAN;
  LOCAL
    types : SET OF STRING := stripped_typeof (spc);
  END_LOCAL;
  IF 'FINITE_SPACE' IN types THEN
    RETURN (bool(SIZEOF (spc\finite_space.members) = 1));
  END_IF;
  IF 'FINITE_INTEGER_INTERVAL' IN types THEN
    RETURN (bool(spc\finite_integer_interval.size = 1));
  END_IF;
  RETURN (FALSE);
END_FUNCTION;  -- space_is_singleton
(*
```

© ISO 2002 — All rights reserved

Argument definitions:

spc: (input) The space to be examined.

return: (output) The BOOLEAN value indicating whether the space has a single member or not.

4.6.144 stripped_typeof

This function takes the set of strings produced by the **TYPEOF** function from its generic argument and strips **schema_prefix** from each string which begins with that substring.

EXPRESS specification:

```
*)
FUNCTION stripped_typeof(arg : GENERIC:G) : SET OF STRING;
  LOCAL
    types : SET OF STRING := TYPEOF (arg);
    stypes : SET OF STRING := [];
    n : INTEGER := LENGTH (schema_prefix);
  END_LOCAL;
  REPEAT i := 1 TO SIZEOF (types);
    IF types[i][1:n] = schema_prefix THEN
       stypes := stypes + [types[i][n+1:LENGTH(types[i])]];
    ELSE
       stypes := stypes + [types[i]];
    END_IF;
  END_REPEAT;
  RETURN (stypes);
END_FUNCTION;  -- stripped_typeof
(*
```

Argument definitions:

arg: (input) The value whose set of stripped type names is to be produced.

return: (output) The set of strings corresponding to **TYPEOF(arg)**, but with any prefixed copies of **schema_prefix** removed.

4.6.145 subspace_of

This function attempts to determine whether a given mathematical space is a subspace of another mathematical space. Where the information accessible to this function is not sufficient to determine the matter, UNKNOWN is returned.

EXPRESS specification:

```
*)
FUNCTION subspace_of(space1, space2 : maths_space) : LOGICAL;
  LOCAL
    spc1 : maths_space := simplify_maths_space(space1);
    spc2 : maths_space := simplify_maths_space(space2);
```

 © ISO 2002 — All rights reserved

```
    types1 : SET OF STRING := stripped_typeof (spc1);
    types2 : SET OF STRING := stripped_typeof (spc2);
    lgcl, cum : LOGICAL;
    es_val : elementary_space_enumerators;
    bnd1, bnd2 : REAL;
    n : INTEGER;
    sp1, sp2 : maths_space;
    prgn1, prgn2 : polar_complex_number_region;
    aitv : finite_real_interval;
  END_LOCAL;
  IF NOT EXISTS (spc1) OR NOT EXISTS (spc2) THEN
    RETURN (FALSE);
  END_IF;
  IF spc2 = the_generics THEN
    RETURN (TRUE);
  END_IF;
  IF 'ELEMENTARY_SPACE' IN types1 THEN
    IF NOT ('ELEMENTARY_SPACE' IN types2) THEN
      RETURN (FALSE);
    END_IF;
    es_val := spc2\elementary_space.space_id;
    IF spc1\elementary_space.space_id = es_val THEN
      RETURN (TRUE);
    END_IF;
    -- Note that the cases (spc2=the_generics) and (spc1=spc2) have been handled.
    CASE spc1\elementary_space.space_id OF
    es_numbers :  RETURN (FALSE);
    es_complex_numbers :  RETURN (es_val = es_numbers);
    es_reals :  RETURN (es_val = es_numbers);
    es_integers :  RETURN (es_val = es_numbers);
    es_logicals :  RETURN (FALSE);
    es_booleans :  RETURN (es_val = es_logicals);
    es_strings :  RETURN (FALSE);
    es_binarys :  RETURN (FALSE);
    es_maths_spaces :  RETURN (FALSE);
    es_maths_functions :  RETURN (FALSE);
    es_generics :  RETURN (FALSE);
    END_CASE;
    -- Should be unreachable.
    RETURN (UNKNOWN);
  END_IF;
  IF 'FINITE_INTEGER_INTERVAL' IN types1 THEN
    cum := TRUE;
    REPEAT i := spc1\finite_integer_interval.min TO spc1\finite_integer_interval.max;
      cum := cum AND member_of (i, spc2);
      IF cum = FALSE THEN
        RETURN (FALSE);
      END_IF;
    END_REPEAT;
    RETURN (cum);
  END_IF;
  IF 'INTEGER_INTERVAL_FROM_MIN' IN types1 THEN
    IF 'ELEMENTARY_SPACE' IN types2 THEN
      es_val := spc2\elementary_space.space_id;
      RETURN ((es_val = es_numbers) OR (es_val = es_integers));
    END_IF;
    IF 'INTEGER_INTERVAL_FROM_MIN' IN types2 THEN
      RETURN (spc1\integer_interval_from_min.min>=spc2\integer_interval_from_min.min);
    END_IF;
```

© ISO 2002 — All rights reserved

```
    RETURN (FALSE);
  END_IF;
  IF 'INTEGER_INTERVAL_TO_MAX' IN types1 THEN
    IF 'ELEMENTARY_SPACE' IN types2 THEN
      es_val := spc2\elementary_space.space_id;
      RETURN ((es_val = es_numbers) OR (es_val = es_integers));
    END_IF;
    IF 'INTEGER_INTERVAL_TO_MAX' IN types2 THEN
      RETURN (spc1\integer_interval_to_max.max <= spc2\integer_interval_to_max.max);
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'FINITE_REAL_INTERVAL' IN types1 THEN
    IF 'ELEMENTARY_SPACE' IN types2 THEN
      es_val := spc2\elementary_space.space_id;
      RETURN ((es_val = es_numbers) OR (es_val = es_reals));
    END_IF;
    IF ('FINITE_REAL_INTERVAL' IN types2) OR
      ('REAL_INTERVAL_FROM_MIN' IN types2) OR
      ('REAL_INTERVAL_TO_MAX' IN types2) THEN
      IF min_exists (spc2) THEN
        bnd1 := spc1\finite_real_interval.min;
        bnd2 := real_min (spc2);
        IF (bnd1 < bnd2) OR ((bnd1 = bnd2) AND min_included (spc1) AND NOT
          min_included (spc2)) THEN
          RETURN (FALSE);
        END_IF;
      END_IF;
      IF max_exists (spc2) THEN
        bnd1 := spc1\finite_real_interval.max;
        bnd2 := real_max (spc2);
        IF (bnd1 > bnd2) OR ((bnd1 = bnd2) AND max_included (spc1) AND NOT
          max_included (spc2)) THEN
          RETURN (FALSE);
        END_IF;
      END_IF;
      RETURN (TRUE);
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'REAL_INTERVAL_FROM_MIN' IN types1 THEN
    IF 'ELEMENTARY_SPACE' IN types2 THEN
      es_val := spc2\elementary_space.space_id;
      RETURN ((es_val = es_numbers) OR (es_val = es_reals));
    END_IF;
    IF 'REAL_INTERVAL_FROM_MIN' IN types2 THEN
      bnd1 := spc1\real_interval_from_min.min;
      bnd2 := spc2\real_interval_from_min.min;
      RETURN ((bnd2 < bnd1) OR ((bnd2 = bnd1) AND (min_included (spc2) OR
        NOT min_included (spc1))));
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'REAL_INTERVAL_TO_MAX' IN types1 THEN
    IF 'ELEMENTARY_SPACE' IN types2 THEN
      es_val := spc2\elementary_space.space_id;
      RETURN ((es_val = es_numbers) OR (es_val = es_reals));
    END_IF;
    IF 'REAL_INTERVAL_TO_MAX' IN types2 THEN
```

 © ISO 2002 — All rights reserved

```
    bnd1 := spc1\real_interval_to_max.max;
    bnd2 := spc2\real_interval_to_max.max;
    RETURN ((bnd2 > bnd1) OR ((bnd2 = bnd1) AND (max_included (spc2) OR
      NOT max_included (spc1))));
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'CARTESIAN_COMPLEX_NUMBER_REGION' IN types1 THEN
  IF 'ELEMENTARY_SPACE' IN types2 THEN
    es_val := spc2\elementary_space.space_id;
    RETURN ((es_val = es_numbers) OR (es_val = es_complex_numbers));
  END_IF;
  IF 'CARTESIAN_COMPLEX_NUMBER_REGION' IN types2 THEN
    RETURN (subspace_of(spc1\cartesian_complex_number_region.real_constraint,
      spc2\cartesian_complex_number_region.real_constraint) AND
      subspace_of(spc1\cartesian_complex_number_region.imag_constraint,
      spc2\cartesian_complex_number_region.imag_constraint));
  END_IF;
  IF 'POLAR_COMPLEX_NUMBER_REGION' IN types2 THEN
    RETURN (subspace_of(enclose_cregion_in_pregion(spc1,
      spc2\polar_complex_number_region.centre),spc2));
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'POLAR_COMPLEX_NUMBER_REGION' IN types1 THEN
  IF 'ELEMENTARY_SPACE' IN types2 THEN
    es_val := spc2\elementary_space.space_id;
    RETURN ((es_val = es_numbers) OR (es_val = es_complex_numbers));
  END_IF;
  IF 'CARTESIAN_COMPLEX_NUMBER_REGION' IN types2 THEN
    RETURN (subspace_of(enclose_pregion_in_cregion(spc1),spc2));
  END_IF;
  IF 'POLAR_COMPLEX_NUMBER_REGION' IN types2 THEN
    prgn1 := spc1;
    prgn2 := spc2;
    IF prgn1.centre = prgn2.centre THEN
      IF prgn2.direction_constraint.max > PI THEN
        aitv := make_finite_real_interval(-PI,open,prgn2.direction_constraint.max
          -2.0*PI,prgn2.direction_constraint.max_closure);
        RETURN (subspace_of(prgn1.distance_constraint,prgn2.distance_constraint)
          AND (subspace_of(prgn1.direction_constraint,prgn2.direction_constraint)
            OR subspace_of(prgn1.direction_constraint,aitv)));
      ELSE
        RETURN (subspace_of(prgn1.distance_constraint,prgn2.distance_constraint)
          AND subspace_of(prgn1.direction_constraint,prgn2.direction_constraint));
      END_IF;
    END_IF;
    RETURN (subspace_of(enclose_pregion_in_pregion(prgn1,prgn2.centre),prgn2));
  END_IF;
  RETURN (FALSE);
END_IF;
IF 'FINITE_SPACE' IN types1 THEN
  cum := TRUE;
  REPEAT i := 1 TO SIZEOF (spc1\finite_space.members);
    cum := cum AND member_of (spc1\finite_space.members[i], spc2);
    IF cum = FALSE THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
```

© ISO 2002 — All rights reserved

```
    RETURN (cum);
  END_IF;
  IF 'PRODUCT_SPACE' IN types1 THEN
    IF 'PRODUCT_SPACE' IN types2 THEN
      IF space_dimension (spc1) = space_dimension (spc2) THEN
        cum := TRUE;
        REPEAT i := 1 TO space_dimension (spc1);
          cum := cum AND subspace_of (factor_space(spc1,i), factor_space(spc2,i));
          IF cum = FALSE THEN
            RETURN (FALSE);
          END_IF;
        END_REPEAT;
        RETURN (cum);
      END_IF;
    END_IF;
    IF 'EXTENDED_TUPLE_SPACE' IN types2 THEN
      IF space_dimension (spc1) >= space_dimension (spc2) THEN
        cum := TRUE;
        REPEAT i := 1 TO space_dimension (spc1);
          cum := cum AND subspace_of (factor_space(spc1,i), factor_space(spc2,i));
          IF cum = FALSE THEN
            RETURN (FALSE);
          END_IF;
        END_REPEAT;
        RETURN (cum);
      END_IF;
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'EXTENDED_TUPLE_SPACE' IN types1 THEN
    IF 'EXTENDED_TUPLE_SPACE' IN types2 THEN
      n := space_dimension (spc1);
      IF n < space_dimension (spc2) THEN
        n := space_dimension (spc2);
      END_IF;
      cum := TRUE;
      REPEAT i := 1 TO n+1;
        cum := cum AND subspace_of (factor_space(spc1,i), factor_space(spc2,i));
        IF cum = FALSE THEN
          RETURN (FALSE);
        END_IF;
      END_REPEAT;
      RETURN (cum);
    END_IF;
    RETURN (FALSE);
  END_IF;
  IF 'FUNCTION_SPACE' IN types1 THEN
    IF 'ELEMENTARY_SPACE' IN types2 THEN
      RETURN (spc2\elementary_space.space_id = es_maths_functions);
    END_IF;
    IF 'FUNCTION_SPACE' IN types2 THEN
      cum := TRUE;
      sp1 := spc1\function_space.domain_argument;
      sp2 := spc2\function_space.domain_argument;
      CASE spc1\function_space.domain_constraint OF
      sc_equal : BEGIN
        CASE spc2\function_space.domain_constraint OF
        sc_equal : cum := cum AND equal_maths_spaces (sp1, sp2);
        sc_subspace : cum := cum AND subspace_of (sp1, sp2);
```

 © ISO 2002 — All rights reserved

```
    sc_member : cum := cum AND member_of (sp1, sp2);
    END_CASE;
    END;
  sc_subspace : BEGIN
    CASE spc2\function_space.domain_constraint OF
    sc_equal : RETURN (FALSE);
    sc_subspace : cum := cum AND subspace_of (sp1, sp2);
    sc_member : BEGIN
      IF NOT member_of (sp1, sp2) THEN
        RETURN (FALSE);
      END_IF;
      cum := UNKNOWN;
      END;
    END_CASE;
    END;
  sc_member : BEGIN
    CASE spc2\function_space.domain_constraint OF
    sc_equal : cum := cum AND space_is_singleton(sp1) AND
      equal_maths_spaces(singleton_member_of(sp1),sp2);
    sc_subspace : BEGIN
      IF NOT member_of (sp2, sp1) THEN
        RETURN (FALSE);
      END_IF;
      cum := UNKNOWN;
      END;
    sc_member : cum := cum AND (subspace_of (sp1, sp2));
    END_CASE;
    END;
  END_CASE;
  IF cum = FALSE THEN
    RETURN (FALSE);
  END_IF;
  sp1 := spc1\function_space.range_argument;
  sp2 := spc2\function_space.range_argument;
  CASE spc1\function_space.range_constraint OF
  sc_equal : BEGIN
    CASE spc2\function_space.range_constraint OF
    sc_equal : cum := cum AND equal_maths_spaces (sp1, sp2);
    sc_subspace : cum := cum AND subspace_of (sp1, sp2);
    sc_member : cum := cum AND member_of (sp1, sp2);
    END_CASE;
    END;
  sc_subspace : BEGIN
    CASE spc2\function_space.domain_constraint OF
    sc_equal : RETURN (FALSE);
    sc_subspace : cum := cum AND subspace_of (sp1, sp2);
    sc_member : BEGIN
      IF NOT member_of (sp1, sp2) THEN
        RETURN (FALSE);
      END_IF;
      cum := UNKNOWN;
      END;
    END_CASE;
    END;
  sc_member : BEGIN
    CASE spc2\function_space.domain_constraint OF
    sc_equal : cum := cum AND space_is_singleton(sp1) AND
      equal_maths_spaces(singleton_member_of(sp1),sp2);
    sc_subspace : BEGIN
```

© ISO 2002 — All rights reserved

```
          IF NOT member_of (sp2, sp1) THEN
            RETURN (FALSE);
          END_IF;
          cum := UNKNOWN;
          END;
        sc_member : cum := cum AND subspace_of (sp1, sp2);
        END_CASE;
        END;
      END_CASE;
      RETURN (cum);
    END_IF;
    RETURN (FALSE);
  END_IF;
  -- Should be unreachable
  RETURN (UNKNOWN);
END_FUNCTION;  -- subspace_of
(*
```

Argument definitions:

space1: (input) The first space.

space2: (input) The second space.

return: (output) A LOGICAL value which is TRUE if the first space can be determined to be a subspace of the second space, FALSE if the first space can be determined not to be a subspace of the second space, and UNKNOWN, otherwise.

4.6.146 subspace_of_es

This function determins whether a **maths_space** is a subspace of an elementary space identified by a value of **elementary_space_enumerators**.

EXPRESS specification:

```
*)
FUNCTION subspace_of_es(spc : maths_space;
                        es  : elementary_space_enumerators) : LOGICAL;
  LOCAL
    types : SET OF STRING := stripped_typeof(spc);
  END_LOCAL;
  IF NOT EXISTS (spc) OR NOT EXISTS (es) THEN  RETURN (FALSE);  END_IF;
  IF 'ELEMENTARY_SPACE' IN types THEN
    RETURN (es_subspace_of_es(spc\elementary_space.space_id,es));
  END_IF;
  IF 'FINITE_SPACE' IN types THEN
    RETURN (all_members_of_es(spc\finite_space.members,es));
  END_IF;
  CASE es OF
  es_numbers : RETURN (
    ('FINITE_INTEGER_INTERVAL' IN types) OR
    ('INTEGER_INTERVAL_FROM_MIN' IN types) OR
    ('INTEGER_INTERVAL_TO_MAX' IN types) OR
    ('FINITE_REAL_INTERVAL' IN types) OR
    ('REAL_INTERVAL_FROM_MIN' IN types) OR
```


© ISO 2002 — All rights reserved

```
    ('REAL_INTERVAL_TO_MAX' IN types) OR
    ('CARTESIAN_COMPLEX_NUMBER_REGION' IN types) OR
    ('POLAR_COMPLEX_NUMBER_REGION' IN types) );
  es_complex_numbers : RETURN (
    ('CARTESIAN_COMPLEX_NUMBER_REGION' IN types) OR
    ('POLAR_COMPLEX_NUMBER_REGION' IN types) );
  es_reals : RETURN (
    ('FINITE_REAL_INTERVAL' IN types) OR
    ('REAL_INTERVAL_FROM_MIN' IN types) OR
    ('REAL_INTERVAL_TO_MAX' IN types) );
  es_integers : RETURN (
    ('FINITE_INTEGER_INTERVAL' IN types) OR
    ('INTEGER_INTERVAL_FROM_MIN' IN types) OR
    ('INTEGER_INTERVAL_TO_MAX' IN types) );
  es_logicals : RETURN (FALSE);
  es_booleans : RETURN (FALSE);
  es_strings : RETURN (FALSE);
  es_binarys : RETURN (FALSE);
  es_maths_spaces : RETURN (FALSE);
  es_maths_functions : RETURN ('FUNCTION_SPACE' IN types);
  es_generics : RETURN (TRUE);
  END_CASE;
  RETURN (UNKNOWN);
END_FUNCTION;  -- subspace_of_es
(*
```

Argument definitions:

spc: (input) The instance of **maths_space** to be tested.

es: (input) The enumeration value identifying the elementary space.

return: (output) A LOGICAL value which is TRUE if the space can be determined to be a subspace of the elementary space, FALSE if the space can be determined not to be a subspace of the elementary space, and UNKNOWN, otherwise.

4.6.147 substitute

This function constructs an expression by substituting values for variables in a given expression. The list of values is substituted for the corresponding variables simultaneously.

NOTE Since the substitution occurs simultaneously, it is possible, for example, to substitute values y and x for variables x and y in the expression $2x(y-x)$ and get $2y(x-y)$.

EXPRESS specification:

```
*)
FUNCTION substitute(expr : generic_expression;
                    vars : LIST [1:?] OF generic_variable;
                    vals : LIST [1:?] OF maths_value) : generic_expression;
  LOCAL
    types : SET OF STRING := stripped_typeof(expr);
    opnds : LIST OF generic_expression;
    op1, op2 : generic_expression;
    qvars : LIST OF generic_variable;
```

© ISO 2002 — All rights reserved

```
    srcdom : maths_space_or_function;
    prpfun : LIST [1:?] OF maths_function;
    finfun : maths_function_select;
  END_LOCAL;
  IF SIZEOF (vars) <> SIZEOF (vals) THEN  RETURN (?);  END_IF;
  IF 'GENERIC_LITERAL' IN types THEN  RETURN (expr);  END_IF;
  IF 'GENERIC_VARIABLE' IN types THEN
    REPEAT i := 1 TO SIZEOF (vars);
      IF expr :=: vars[i] THEN  RETURN (vals[i]);  END_IF;
    END_REPEAT;
    RETURN (expr);
  END_IF;
  IF 'QUANTIFIER_EXPRESSION' IN types THEN
    qvars := expr\quantifier_expression.variables;
    -- Variables subject to a quantifier do not participate in this kind of
    -- substitution process.
    REPEAT i := SIZEOF (vars) TO 1 BY -1;
      IF vars[i] IN qvars THEN
        REMOVE (vars, i);
        REMOVE (vals, i);
      END_IF;
    END_REPEAT;
    opnds := expr\multiple_arity_generic_expression.operands;
    REPEAT i := 1 TO SIZEOF (opnds);
      IF NOT (opnds[i] IN qvars) THEN
        expr\multiple_arity_generic_expression.operands[i] :=
          substitute(opnds[i],vars,vals);
        -- This technique will not work on subtypes of quantifier_expression
        -- which derive their operands from other attributes!
      END_IF;
    END_REPEAT;
    RETURN (expr);  -- operands modified!
  END_IF;
  IF 'UNARY_GENERIC_EXPRESSION' IN types THEN
    op1 := expr\unary_generic_expression.operand;
    expr\unary_generic_expression.operand := substitute(op1, vars, vals);
    -- This technique will not work on subtypes of unary_generic_expression
    -- which derive their operands from other attributes!
  END_IF;
  IF 'BINARY_GENERIC_EXPRESSION' IN types THEN
    op1 := expr\binary_generic_expression.operands[1];
    expr\binary_generic_expression.operands[1] := substitute(op1, vars, vals);
    op2 := expr\binary_generic_expression.operands[2];
    expr\binary_generic_expression.operands[2] := substitute(op2, vars, vals);
    -- This technique will not work on subtypes of binary_generic_expression
    -- which derive their operands from other attributes!
  END_IF;
  IF 'PARALLEL_COMPOSED_FUNCTION' IN types THEN
    -- Subtype of multiple_arity_generic_expression which derives its operands.
    srcdom := expr\parallel_composed_function.source_of_domain;
    prpfun := expr\parallel_composed_function.prep_functions;
    finfun := expr\parallel_composed_function.final_function;
    srcdom := substitute(srcdom,vars,vals);
    REPEAT i := 1 TO SIZEOF (prpfun);
      prpfun[i] := substitute(prpfun[i],vars,vals);
    END_REPEAT;
    IF 'MATHS_FUNCTION' IN stripped_typeof(finfun) THEN
      finfun := substitute(finfun,vars,vals);
    END_IF;
```

 © ISO 2002 — All rights reserved

```
    RETURN (make_parallel_composed_function(srcdom,prpfun,finfun));
  END_IF;
  IF 'MULTIPLE_ARITY_GENERIC_EXPRESSION' IN types THEN
    opnds := expr\multiple_arity_generic_expression.operands;
    REPEAT i := 1 TO SIZEOF (opnds);
      expr\multiple_arity_generic_expression.operands[i] :=
        substitute(opnds[i],vars,vals);
        -- This technique will not work on subtypes of multiple_arity_generic_
        -- expression which derive their operands from other attributes!
    END_REPEAT;
  END_IF;
  RETURN (expr);
END_FUNCTION;  -- substitute
(*
```

Argument definitions:

expr: (input) The expression to be substituted.

vars: (input) The list of variable to be replaced.

vals: (input) The list of mathematical values to substitute for the corresponding variables

return: (output) The expression created by carrying out the substitutions.

4.6.148 values_space_of

This function produces an instance of **maths_space** representing a mathematical space which contains all possible values of the mathematical expression given as input. A possible value of an expression is one which could be the denotation of the expression after some assignment of legitimate values to the free variables in the expression. If the expression is not recognizable as a mathematical expression by the function **has_values_space**, the indeterminate value is returned.

Analogously to the range space of a function, the values space of an expression is only a space containing all the possible values. It may contain many values which are not actual values of the expression for any assignment of values to free variables.

NOTE The *actual* value space of an expression is impractical to ascertain in general. The purpose of this function is that it gives an equivalent of "type" information for mathematical expressions.

EXPRESS specification:

```
*)
FUNCTION values_space_of(expr : generic_expression) : maths_space;
  LOCAL
    e_prefix : STRING := 'ISO13584_EXPRESSIONS_SCHEMA.';
    typenames : SET OF STRING := TYPEOF (expr);
  END_LOCAL;
  IF (schema_prefix + 'MATHS_VARIABLE') IN typenames THEN
    RETURN (expr\maths_variable.values_space);
  END_IF;
  IF (e_prefix + 'EXPRESSION') IN typenames THEN
    IF (e_prefix + 'NUMERIC_EXPRESSION') IN typenames THEN
      IF expr\numeric_expression.is_int THEN
```

© ISO 2002 — All rights reserved

```
      IF (e_prefix + 'INT_LITERAL') IN typenames THEN
        RETURN (make_finite_space ([expr\int_literal.the_value]));
      ELSE
        RETURN (the_integers);
      END_IF;
    ELSE
      IF (e_prefix + 'REAL_LITERAL') IN typenames THEN
        RETURN (make_finite_space ([expr\real_literal.the_value]));
      ELSE
        RETURN (the_reals);
      END_IF;
    END_IF;
  END_IF;
  IF (e_prefix + 'BOOLEAN_EXPRESSION') IN typenames THEN
    IF (e_prefix + 'BOOLEAN_LITERAL') IN typenames THEN
      RETURN (make_finite_space ([expr\boolean_literal.the_value]));
    ELSE
      RETURN (the_booleans);
    END_IF;
  END_IF;
  IF (e_prefix + 'STRING_EXPRESSION') IN typenames THEN
    IF (e_prefix + 'STRING_LITERAL') IN typenames THEN
      RETURN (make_finite_space ([expr\string_literal.the_value]));
    ELSE
      RETURN (the_strings);
    END_IF;
  END_IF;
  RETURN (?);  -- unknown subtype of expression
END_IF;
IF (schema_prefix + 'MATHS_FUNCTION') IN typenames THEN
  IF expression_is_constant (expr) THEN
    RETURN (make_finite_space ([expr]));
  ELSE
    RETURN (make_function_space (sc_equal, expr\maths_function.domain,
      sc_equal, expr\maths_function.range));
  END_IF;
END_IF;
IF (schema_prefix + 'FUNCTION_APPLICATION') IN typenames THEN
  RETURN (expr\function_application.func.range);
END_IF;
IF (schema_prefix + 'MATHS_SPACE') IN typenames THEN
  IF expression_is_constant (expr) THEN
    RETURN (make_finite_space ([expr]));
  ELSE
    -- This case cannot occur in this version of the schema.
    -- When it becomes possible, the subtypes should be analysed and
    -- more finely defined spaces returned.
    RETURN (make_elementary_space (es_maths_spaces));
  END_IF;
END_IF;
IF (schema_prefix + 'DEPENDENT_VARIABLE_DEFINITION') IN typenames THEN
  RETURN (values_space_of (expr\unary_generic_expression.operand));
END_IF;
IF (schema_prefix + 'COMPLEX_NUMBER_LITERAL') IN typenames THEN
  RETURN (make_finite_space ([expr]));
END_IF;
IF (schema_prefix + 'LOGICAL_LITERAL') IN typenames THEN
  RETURN (make_finite_space ([expr\logical_literal.lit_value]));
END_IF;
```

 © ISO 2002 — All rights reserved

```
  IF (schema_prefix + 'BINARY_LITERAL') IN typenames THEN
    RETURN (make_finite_space ([expr\binary_literal.lit_value]));
  END_IF;
  IF (schema_prefix + 'MATHS_ENUM_LITERAL') IN typenames THEN
    RETURN (make_finite_space ([expr\maths_enum_literal.lit_value]));
  END_IF;
  IF (schema_prefix + 'REAL_TUPLE_LITERAL') IN typenames THEN
    RETURN (make_finite_space ([expr\real_tuple_literal.lit_value]));
  END_IF;
  IF (schema_prefix + 'INTEGER_TUPLE_LITERAL') IN typenames THEN
    RETURN (make_finite_space ([expr\integer_tuple_literal.lit_value]));
  END_IF;
  IF (schema_prefix + 'ATOM_BASED_LITERAL') IN typenames THEN
    RETURN (make_finite_space ([expr\atom_based_literal.lit_value]));
  END_IF;
  IF (schema_prefix + 'MATHS_TUPLE_LITERAL') IN typenames THEN
    RETURN (make_finite_space ([expr\maths_tuple_literal.lit_value]));
  END_IF;
  IF (schema_prefix + 'PARTIAL_DERIVATIVE_EXPRESSION') IN typenames THEN
    RETURN (drop_numeric_constraints (values_space_of (
      expr\partial_derivative_expression.derivand)));
  END_IF;
  IF (schema_prefix + 'DEFINITE_INTEGRAL_EXPRESSION') IN typenames THEN
    RETURN (drop_numeric_constraints (values_space_of (
      expr\definite_integral_expression.integrand)));
  END_IF;
  RETURN (?);  -- not recognized as a mathematical expression
END_FUNCTION;  -- values_space_of
(*
```

Argument definitions:

expr: (input) The expression whose value space is to be derived.

return: (output) A space containing all possible values of the expression.

EXPRESS specification:

```
*)
END_SCHEMA;  -- mathematical_functions_schema
(*
```

© ISO 2002 — All rights reserved

Annex A
(normative)

Short names of entities

Table A.1 provides the short names of entities specified in this part of ISO 10303. Requirements on the use of short names are found in the implementation methods included in ISO 10303.

Table A.1 – Short names of entities

Entity names	Short names
ABSTRACTED_EXPRESSION_FUNCTION	ABEXFN
APPLICATION_DEFINED_FUNCTION	APDFFN
ATOM_BASED_LITERAL	ATBSLT
B_SPLINE_BASIS	BSPBS
B_SPLINE_FUNCTION	BSPFN
BANDED_MATRIX	BNDMTR
BASIC_SPARSE_MATRIX	BSSPMT
BINARY_LITERAL	BNRLTR
BOUND_VARIABLE_SEMANTICS	BNVRSM
CARTESIAN_COMPLEX_NUMBER_REGION	CCNR
COMPLEX_NUMBER_LITERAL	CMNMLT
CONSTANT_FUNCTION	CNSFNC
DEFINITE_INTEGRAL_EXPRESSION	DFINEX
DEFINITE_INTEGRAL_FUNCTION	DFINFN
DEPENDENT_VARIABLE_DEFINITION	DPVRDF
ELEMENTARY_FUNCTION	ELMFNC
ELEMENTARY_SPACE	ELMSPC
EXPLICIT_TABLE_FUNCTION	EXTBFN
EXPRESSION_DENOTED_FUNCTION	EXDNFN
EXTENDED_TUPLE_SPACE	EXTPSP
EXTERNALLY_LISTED_DATA	EXLSDT
FINITE_FUNCTION	FNTFNC
FINITE_INTEGER_INTERVAL	FNININ
FINITE_REAL_INTERVAL	FNRLIN

 © ISO 2002 — All rights reserved

Table A.1 – (continued)

Entity names	Short names
FINITE_SPACE	FNTSPC
FREE_VARIABLE_SEMANTICS	FRVRSM
FUNCTION_APPLICATION	FNCAPP
FUNCTION_SPACE	FNCSPC
GENERAL_LINEAR_FUNCTION	GNLNFN
HOMOGENEOUS_LINEAR_FUNCTION	HMLNFN
IMPORTED_CURVE_FUNCTION	IMCRFN
IMPORTED_POINT_FUNCTION	IMPNFN
IMPORTED_SURFACE_FUNCTION	IMSRFN
IMPORTED_VOLUME_FUNCTION	IMVLFN
INTEGER_INTERVAL_FROM_MIN	IIFM
INTEGER_INTERVAL_TO_MAX	IITM
INTEGER_TUPLE_LITERAL	INTPLT
LINEARIZED_TABLE_FUNCTION	LNTBFN
LISTED_COMPLEX_NUMBER_DATA	LCND
LISTED_DATA	LSTDT
LISTED_INTEGER_DATA	LSINDT
LISTED_LOGICAL_DATA	LSLGDT
LISTED_PRODUCT_SPACE	LSPRSP
LISTED_REAL_DATA	LSRLDT
LISTED_STRING_DATA	LSSTDT
LOGICAL_LITERAL	LGCLTR
MATHEMATICAL_DESCRIPTION	MTHDSC
MATHS_BOOLEAN_VARIABLE	MTBLVR
MATHS_ENUM_LITERAL	MTENLT
MATHS_FUNCTION	MTH0
MATHS_INTEGER_VARIABLE	MTINVR
MATHS_REAL_VARIABLE	MTRLVR

© ISO 2002 — All rights reserved

Table A.1 – (concluded)

Entity names	Short names
MATHS_SPACE	MTH1
MATHS_STRING_VARIABLE	MTSTVR
MATHS_TUPLE_LITERAL	MTTPLT
MATHS_VARIABLE	MTHVRB
PARALLEL_COMPOSED_FUNCTION	PRCMFN
PARTIAL_DERIVATIVE_EXPRESSION	PRDREX
PARTIAL_DERIVATIVE_FUNCTION	PRDRFN
POLAR_COMPLEX_NUMBER_REGION	PCNR
QUANTIFIER_EXPRESSION	QNTEXP
RATIONALIZE_FUNCTION	RTNFNC
REAL_INTERVAL_FROM_MIN	RIFM
REAL_INTERVAL_TO_MAX	RITM
REAL_TUPLE_LITERAL	RLTPLT
REGULAR_TABLE_FUNCTION	RGTBFN
REINDEXED_ARRAY_FUNCTION	RNARFN
REPACKAGING_FUNCTION	RPCFNC
RESTRICTION_FUNCTION	RSTFNC
SELECTOR_FUNCTION	SLC0
SERIES_COMPOSED_FUNCTION	SRCMFN
STANDARD_TABLE_FUNCTION	STTBFN
STRICT_TRIANGULAR_MATRIX	STTRMT
SYMMETRIC_BANDED_MATRIX	SYBNMT
SYMMETRIC_MATRIX	SYMMTR
TRIANGULAR_MATRIX	TRNMTR
UNIFORM_PRODUCT_SPACE	UNPRSP

© ISO 2002 — All rights reserved

Annex B
(normative)

Information object registration

B.1 Document identification

To provide for unambiguous identification of an information object in an open system, the object identifier

{ iso standard 10303 part(50) version(1) }

is assigned to this part of ISO 10303. The meaning of this value is defined in ISO/IEC 8824-1, and is described in ISO 10303-1.

B.2 Schema identification

To provide for unambiguous identification of the schema-name in an open information system, the object identifier

{ iso standard 10303 part(50) version(1) schema(1) mathematical-functions-schema(1) }

is assigned to the **mathematical_functions_schema** schema (see clause 4). The meaning of this value is defined in ISO/IEC 8824-1, and is described in ISO 10303-1.

© ISO 2002 — All rights reserved

Annex C
(informative)

Computer-interpretable listings

This annex references a listing of the EXPRESS entity names and corresponding short names as specified in this part of ISO 10303. It also references a listing of each EXPRESS schema specified in this part of ISO 10303, without comments or other explanatory text. These listings are available in computer-interpretable form and can be found at the following URLs:

Short names: `<http://www.mel.nist.gov/div826/subject/apde/snr/>`
EXPRESS: `<http://www.mel.nist.gov/step/parts/part050/is/>`

If there is difficulty accessing these sites contact ISO Central Secretariat or contact the ISO TC 184/SC4 Secretariat directly at: `sc4sec@cme.nist.gov`.

NOTE The information provided in computer-interpretable form at the above URLs is informative. The information that is contained in the body of this part of ISO 10303 is normative.

© ISO 2002 — All rights reserved

Annex D
(informative)

EXPRESS-G diagrams

The diagrams in this annex correspond to the EXPRESS schema specified in this part of ISO 10303. The diagrams use the EXPRESS-G graphical notation for the EXPRESS language. EXPRESS-G is defined in annex D of ISO 10303-11.

© ISO 2002 — All rights reserved

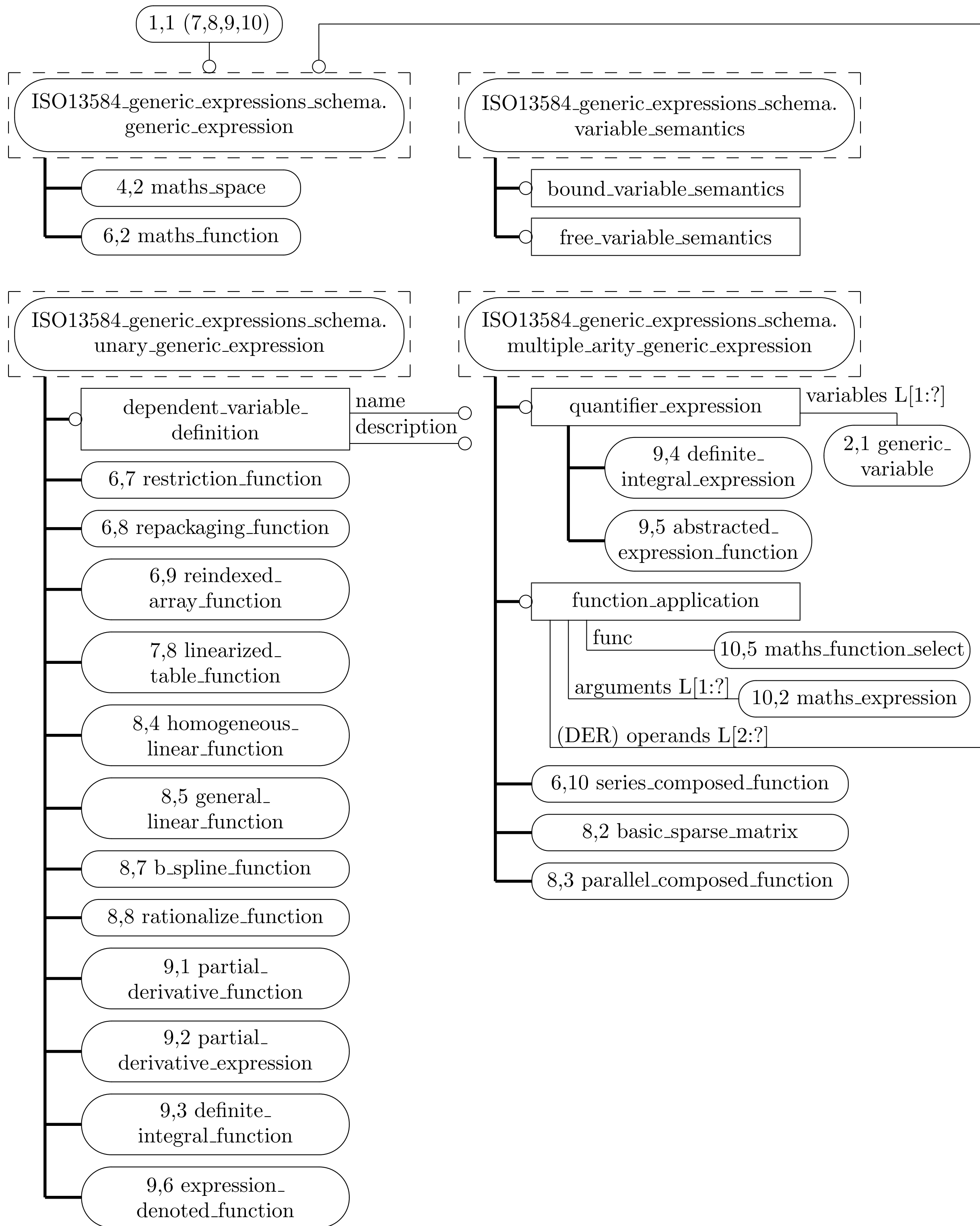

Figure D.1 – EXPRESS-G diagram of the mathematical_functions_schema (1 of 10)

 © ISO 2002 — All rights reserved

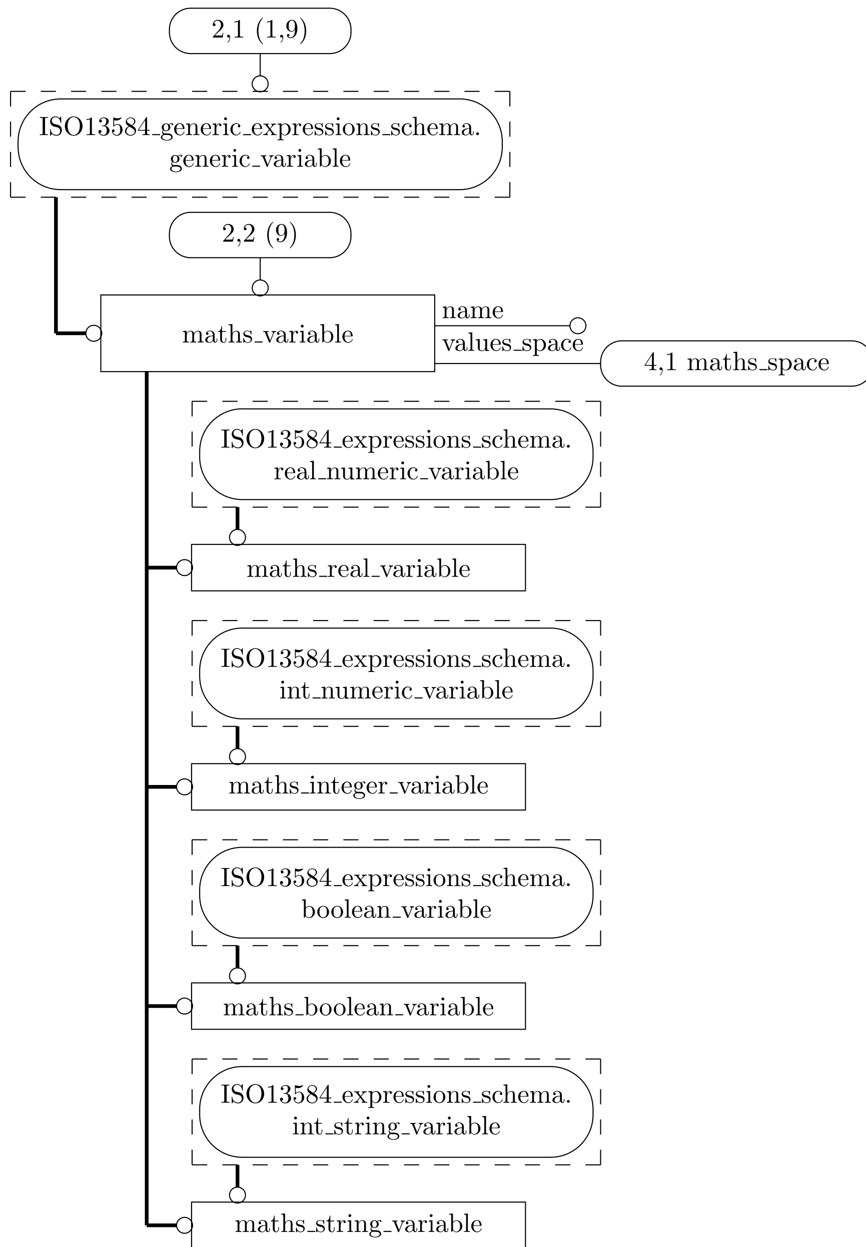

Figure D.2 – EXPRESS-G diagram of the mathematical_functions_schema (2 of 10)

© ISO 2002 — All rights reserved

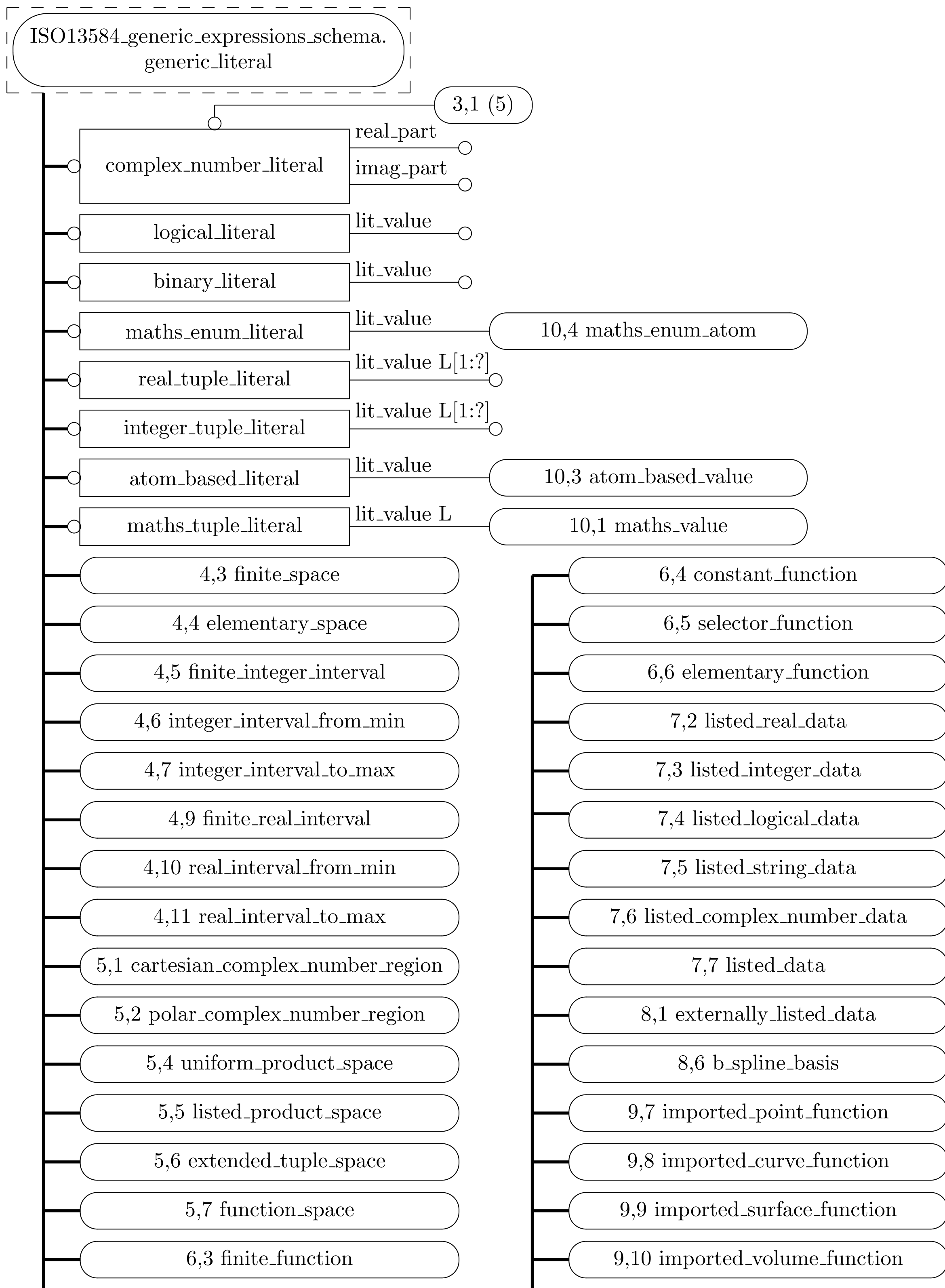

Figure D.3 – EXPRESS-G diagram of the mathematical_functions_schema (3 of 10)

 © ISO 2002 — All rights reserved

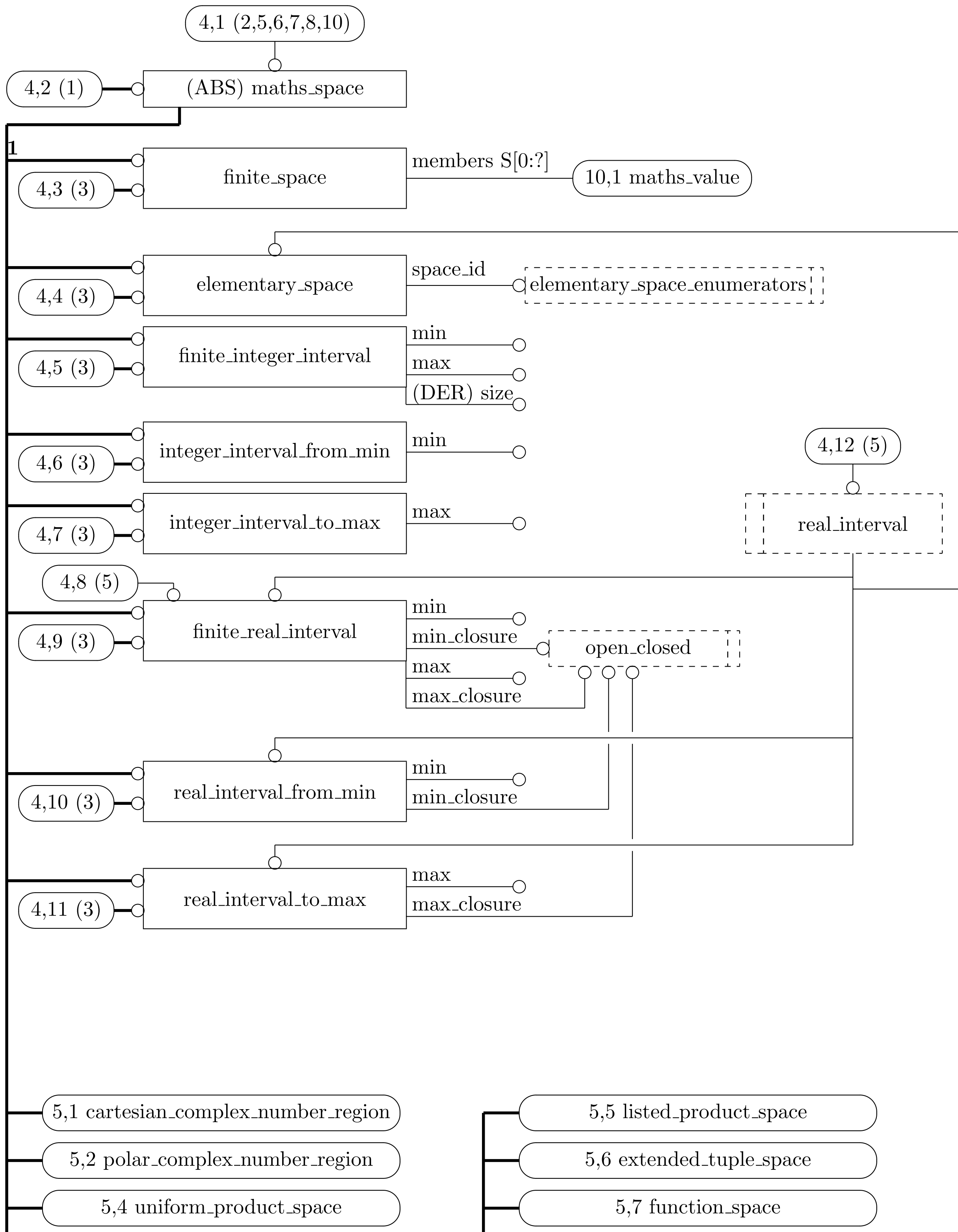

Figure D.4 – EXPRESS-G diagram of the mathematical_functions_schema (4 of 10)

© ISO 2002 — All rights reserved

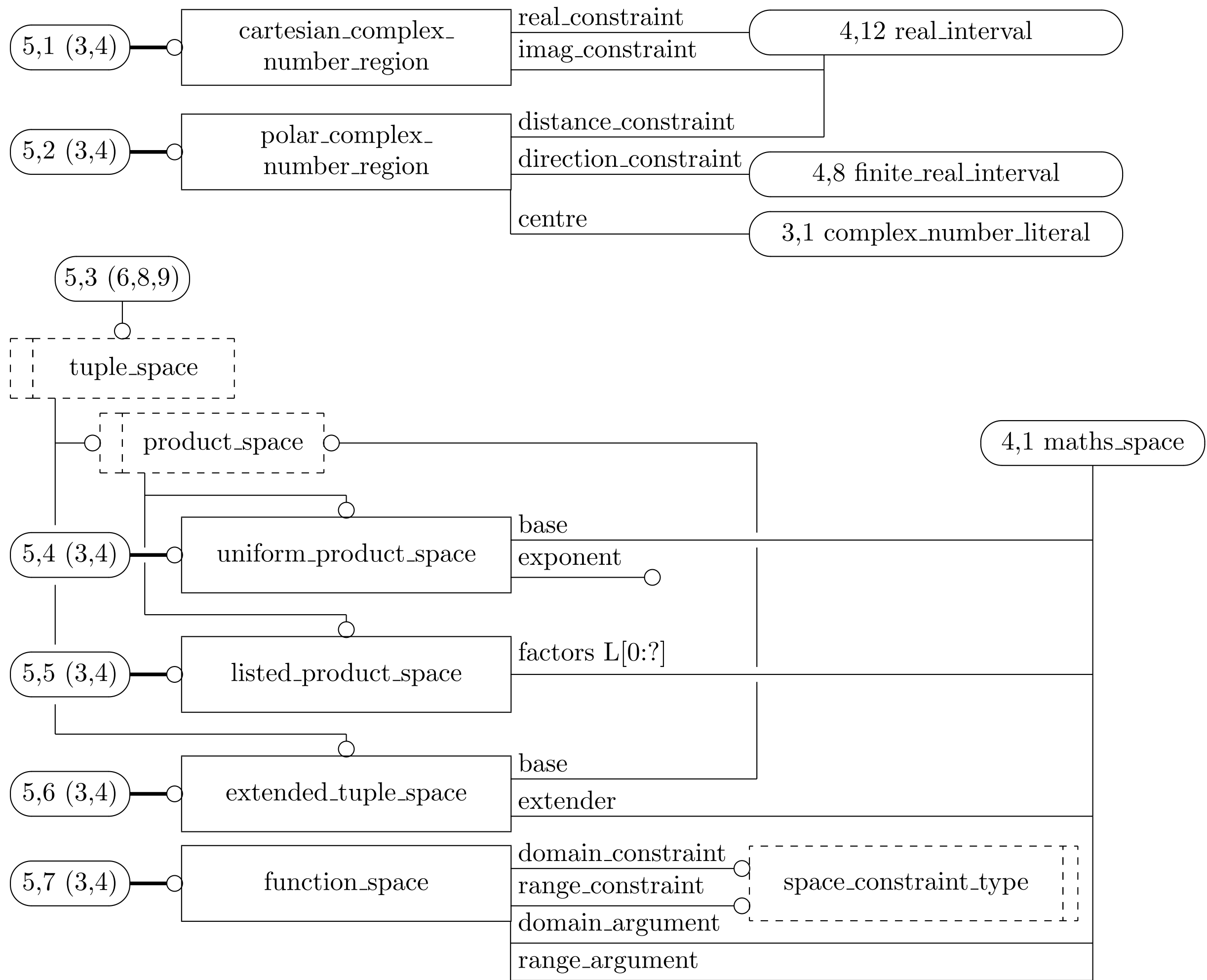

Figure D.5 – EXPRESS-G diagram of the mathematical_functions_schema (5 of 10)

 © ISO 2002 — All rights reserved

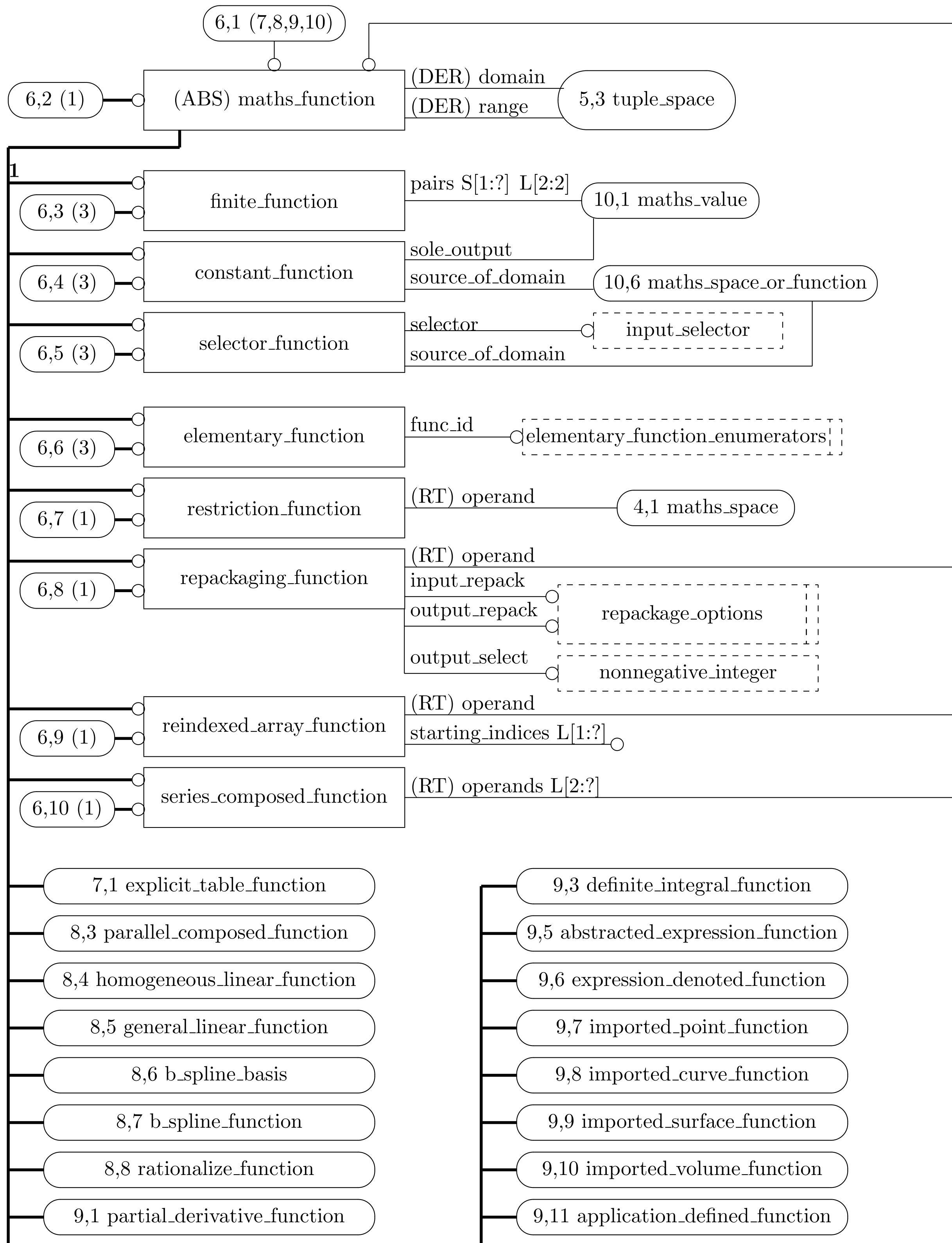

Figure D.6 – EXPRESS-G diagram of the mathematical_functions_schema (6 of 10)

© ISO 2002 — All rights reserved

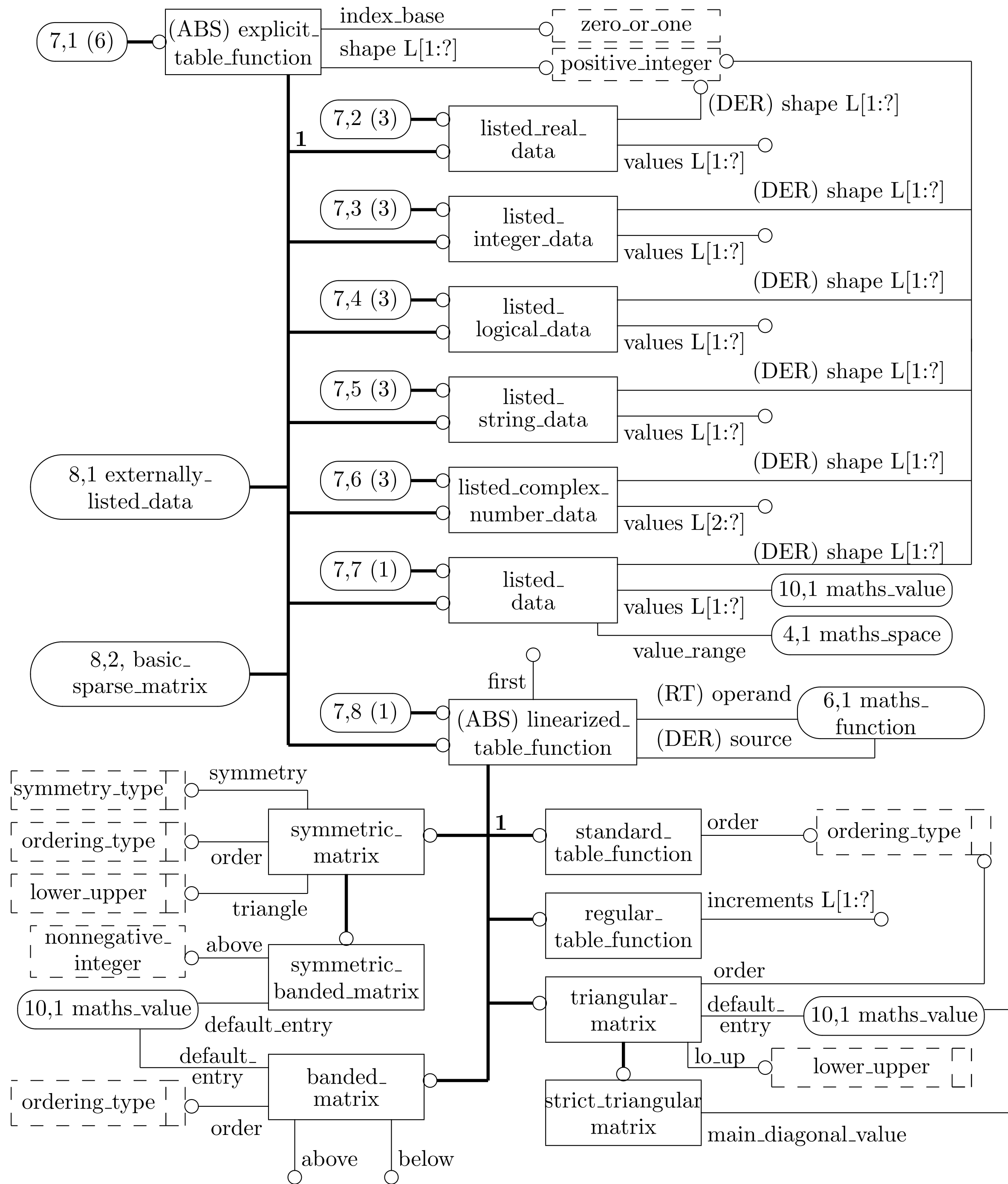

Figure D.7 – EXPRESS-G diagram of the mathematical_functions_schema (7 of 10)

 © ISO 2002 — All rights reserved

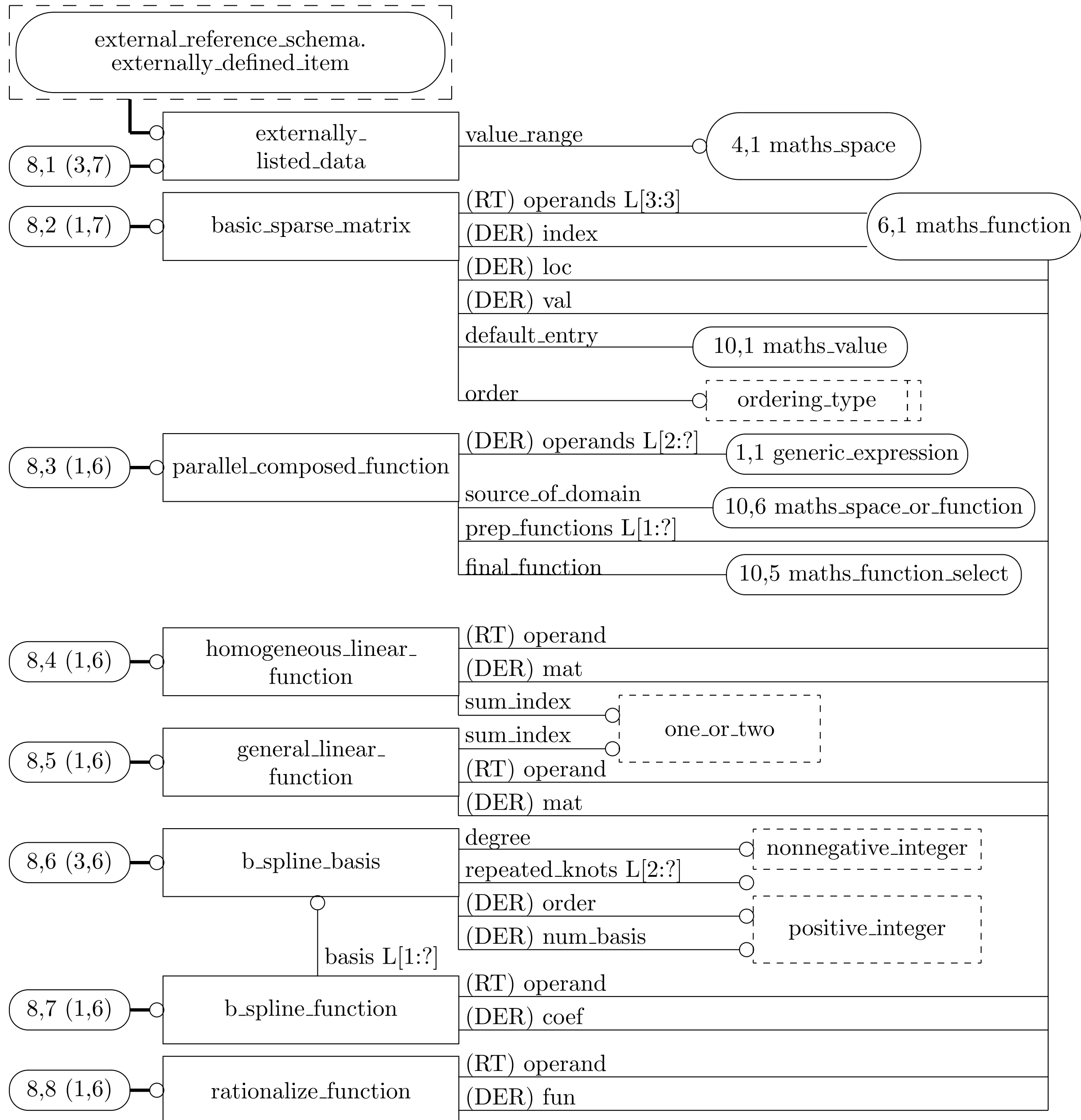

Figure D.8 – EXPRESS-G diagram of the mathematical_functions_schema (8 of 10)

© ISO 2002 — All rights reserved

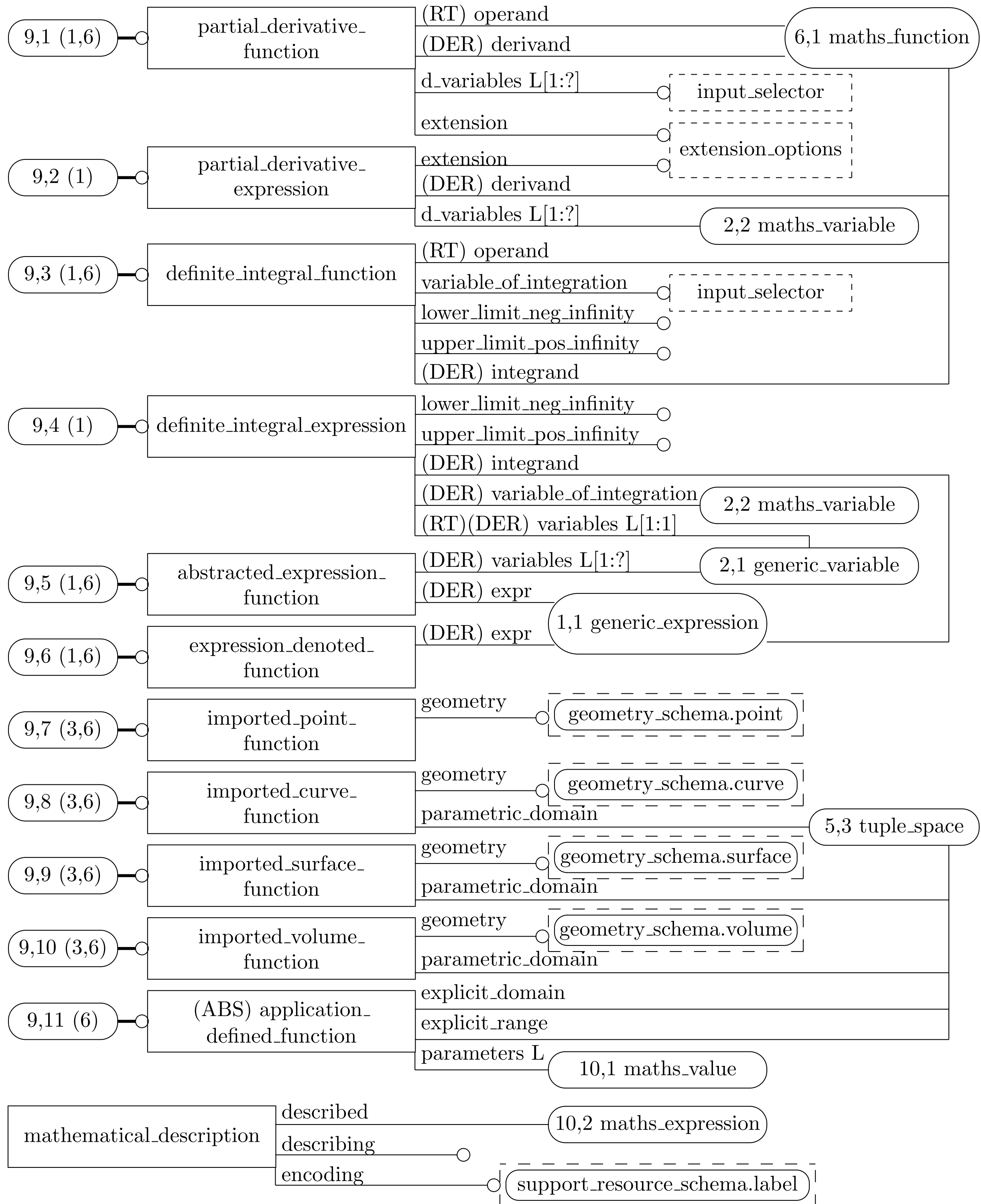

Figure D.9 – EXPRESS-G diagram of the mathematical_functions_schema (9 of 10)

 © ISO 2002 — All rights reserved

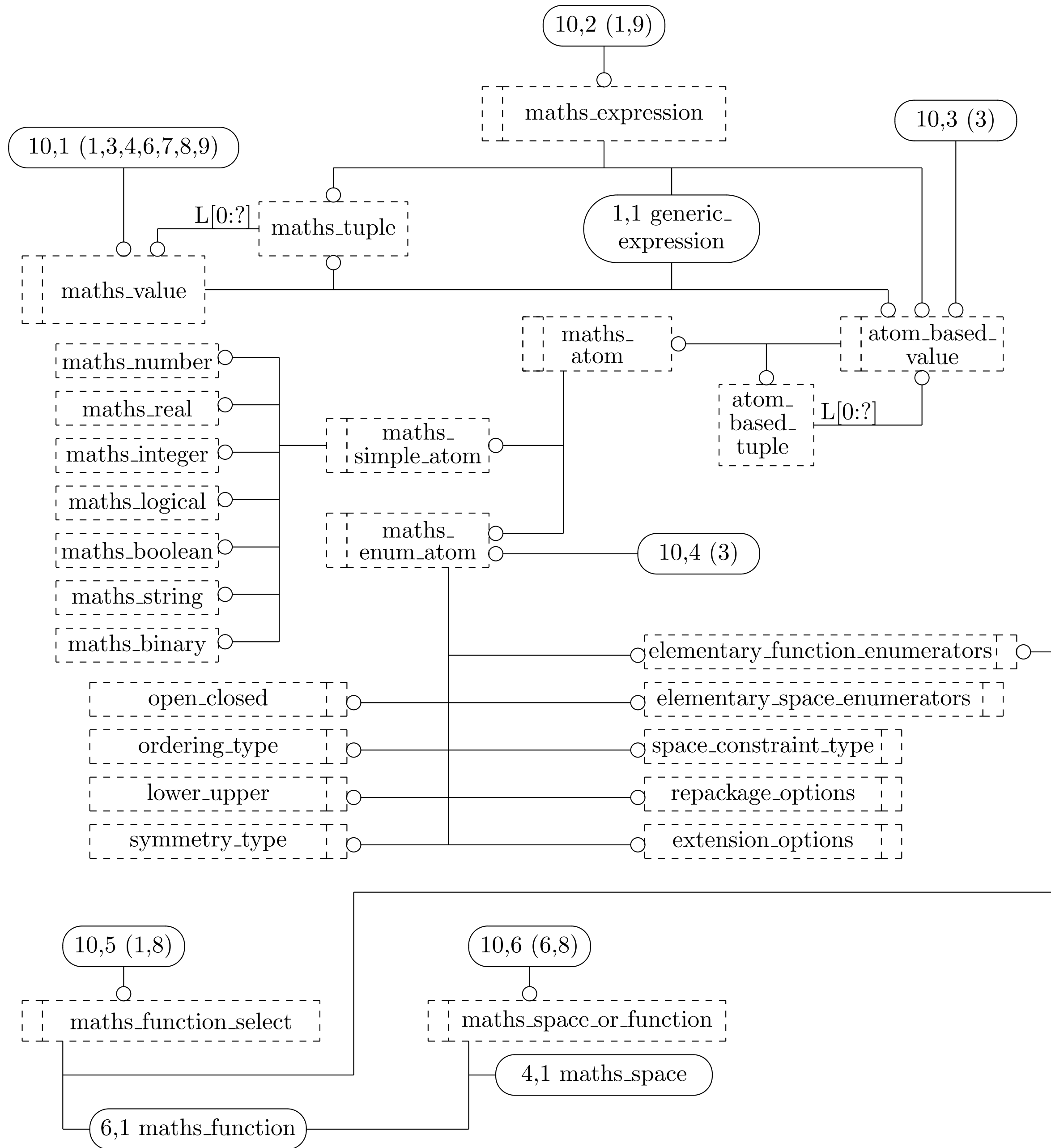

Figure D.10 – EXPRESS-G diagram of the mathematical_functions_schema (10 of 10)

© ISO 2002 — All rights reserved

Bibliography

[1] ANS US PRO/IPO-100-1996. IGES 5.3, *Initial Graphics Exchange Specification,* Sep. 1996.

[2] W3C PR-math-19980224, *Mathematical Markup Language, W3C Proposed Recommendation* [cited 2001-08-13]. Available from the World Wide Web: < http://www.w3.org/TR/1998/PR-math-19980224 > .

[3] W3C REC-MathML-19980407; revised 19990707, *Mathematical Markup Language (MathML) 1.01 Specification* [cited 2001-08-13]. Available from the World Wide Web: < http://www.w3.org /TR/REC-MathML > .

[4] BARENDREGT, H. P. *The Lambda Calculus, Volume 103, Studies in Logic.* Amsterdam: North Holland, 1985.

[5] JAMES, G.; JAMES, R. *Mathematics Dictionary,* third edition, Princeton: D. Van Nostrand Co., Inc., 1968.

© ISO 2002 — All rights reserved

Index

abstracted_expression_function entity ... 97
actual function domain definition ... 2
actual function range definition ... 2
all_members_of_es function ... 104
any_space_satisfies function ... 106
application_defined_function entity ... 102
array function definition ... 2
assoc_product_space function ... 107
atan2 function ... 109
atom_based_literal entity ... 41
atom_based_tuple type ... 19
atom_based_value type ... 19
b_spline_basis entity ... 87
b_spline_function entity ... 88
banded_matrix entity ... 81
basic_sparse_matrix entity ... 83
binary_literal entity ... 39
bool function ... 110
bound variable definition ... 2
bound_variable_semantics entity ... 37
Cartesian product space definition ... 3
cartesian_complex_number_region entity ... 49
check_sparse_index_domain function ... 110
check_sparse_index_to_loc function ... 112
check_sparse_loc_range function ... 111
compare_basis_and_coef function ... 113
compare_list_and_value function ... 113
compare_values function ... 114
compatible spaces definition ... 3
compatible_complex_number_regions function ... 115
compatible_es_values function ... 117
compatible_intervals function ... 118
compatible_spaces function ... 118
complex_number_literal entity ... 38
composable_sequence function ... 124
computable function definition ... 3
constant_function entity ... 58
convert_to_literal function ... 124
convert_to_maths_function function ... 125
convert_to_maths_value function ... 125
convert_to_operand function ... 126
convert_to_operands function ... 127
convert_to_operands_prcmfn function ... 128
definite_integral_check function ... 128
definite_integral_expr_check function ... 129
definite_integral_expression entity ... 95
definite_integral_function entity ... 94
dependent_variable_definition entity ... 37
derive_definite_integral_domain function ... 130
derive_elementary_function_domain function ... 132
derive_elementary_function_range function ... 135

© ISO 2002 — All rights reserved

derive_finite_function_domain function ... 137
derive_finite_function_range function ... 138
derive_function_domain function ... 138
derive_function_range function ... 141
domain_from function ... 144
dot_count function ... 145
dotted_express_identifier type ... 34
dotted_identifiers_syntax function ... 145
drop_numeric_constraints function ... 146
elementary_function entity ... 60
elementary_function_enumerators type ... 26
elementary_space entity ... 46
elementary_space_enumerators type ... 22
enclose_cregion_in_pregion function ... 147
enclose_pregion_in_cregion function ... 151
enclose_pregion_in_pregion function ... 154
equal_cregion_pregion function ... 160
equal_maths_functions function ... 162
equal_maths_spaces function ... 163
equal_maths_values function ... 166
es_subspace_of_es function ... 168
explicit_table_function entity ... 68
express_identifier type ... 34
expression definition ... 3
expression_denoted_function entity ... 98
expression_is_constant function ... 169
extended_tuple_space entity ... 54
extension_options type ... 32
externally_listed_data entity ... 73
extract_factors function ... 169
extremal_position_check function ... 170
factor1 function ... 171
factor_space function ... 172
finite_function entity ... 57
finite_integer_interval entity ... 46
finite_real_interval entity ... 48
finite_space entity ... 51
free variable definition ... 3
free_variable_semantics entity ... 38
free_variables_of function ... 172
function abstraction definition ... 3
function application definition ... 4
function domain definition ... 4
function evaluation definition ... 4
function range definition ... 4
function_applicability function ... 174
function_application entity ... 44
function_is_1d_array function ... 174
function_is_1d_table function ... 175
function_is_2d_table function ... 176
function_is_array function ... 177
function_is_table function ... 177

 © ISO 2002 — All rights reserved

function_space entity 55
general_linear_function entity 86
has_values_space function 178
homogeneous_linear_function entity 85
imported_curve_function entity 100
imported_point_function entity 99
imported_surface_function entity 100
imported_volume_function entity 101
input_selector type 21
integer_interval_from_min entity 47
integer_interval_to_max entity 47
integer_tuple_literal entity 40
linearized_table_function entity 74
list_selected_components function 180
listed_complex_number_data entity 71
listed_data entity 72
listed_integer_data entity 69
listed_logical_data entity 70
listed_product_space entity 53
listed_real_data entity 68
listed_string_data entity 70
logical_literal entity 39
lower_upper type 24
make_abstracted_expression_function function 180
make_atom_based_literal function 181
make_b_spline_basis function 181
make_b_spline_function function 182
make_banded_matrix function 182
make_basic_sparse_matrix function 183
make_binary_literal function 184
make_boolean_literal function 184
make_cartesian_complex_number_region function 185
make_complex_number_literal function 185
make_constant_function function 186
make_cos_expression function 186
make_definite_integral_expression function 187
make_definite_integral_function function 187
make_elementary_function function 188
make_elementary_space function 188
make_environment function 189
make_expression_denoted_function function 189
make_extended_tuple_space function 190
make_finite_function function 190
make_finite_integer_interval function 191
make_finite_real_interval function 191
make_finite_space function 192
make_function_application function 192
make_function_space function 193
make_general_linear_function function 194
make_int_literal function 194
make_integer_interval_from_min function 195
make_listed_complex_number_data function 195

© ISO 2002 — All rights reserved

make_listed_data function . . . 196
make_listed_integer_data function . . . 196
make_listed_product_space function . . . 197
make_listed_real_data function . . . 197
make_logical_literal function . . . 198
make_maths_enum_literal function . . . 198
make_maths_real_variable function . . . 199
make_maths_tuple_literal function . . . 199
make_mult_expression function . . . 200
make_parallel_composed_function function . . . 200
make_partial_derivative_expression function . . . 201
make_partial_derivative_function function . . . 201
make_polar_complex_number_region function . . . 202
make_rationalize_function function . . . 202
make_real_interval_from_min function . . . 203
make_real_interval_to_max function . . . 203
make_real_literal function . . . 204
make_regular_table_function function . . . 204
make_reindexed_array_function function . . . 205
make_repackaging_function function . . . 205
make_selector_function function . . . 206
make_series_composed_function function . . . 207
make_sin_expression function . . . 207
make_standard_table_function function . . . 208
make_strict_triangular_matrix function . . . 208
make_string_literal function . . . 209
make_unary_minus_expression function . . . 210
make_uniform_product_space function . . . 210
mathematical function definition . . . 4
mathematical object definition . . . 4
mathematical set definition . . . 5
mathematical space definition . . . 5
mathematical value definition . . . 4
mathematical_description entity . . . 104
maths_atom type . . . 19
maths_binary type . . . 18
maths_boolean type . . . 18
maths_boolean_variable entity . . . 43
maths_enum_atom type . . . 33
maths_enum_literal entity . . . 39
maths_expression type . . . 20
maths_function entity . . . 56
maths_function_select type . . . 21
maths_integer type . . . 18
maths_integer_variable entity . . . 42
maths_logical type . . . 18
maths_number type . . . 18
maths_real type . . . 18
maths_real_variable entity . . . 42
maths_simple_atom type . . . 18
maths_space entity . . . 45
maths_space_or_function type . . . 35

 © ISO 2002 — All rights reserved

maths_string type ... 18
maths_string_variable entity ... 43
maths_tuple type ... 20
maths_tuple_literal entity ... 41
maths_value type ... 20
maths_variable entity ... 42
matrix definition ... 5
max_exists function ... 211
max_included function ... 211
member_of function ... 212
min_exists function ... 217
min_included function ... 217
no_cyclic_domain_reference function ... 218
no_cyclic_space_reference function ... 219
nondecreasing function ... 220
nonnegative_integer type ... 17
number_superspace_of function ... 220
number_tuple_subspace_check function ... 221
one_or_two type ... 18
one_tuples_of function ... 221
open_closed type ... 31
ordering_type type ... 23
parallel_composed_function entity ... 65
parallel_composed_function_composability_check function ... 222
parallel_composed_function_domain_check function ... 222
parse_express_identifier function ... 223
partial_derivative_check function ... 224
partial_derivative_expression entity ... 92
partial_derivative_function entity ... 91
polar_complex_number_region entity ... 50
positive_integer type ... 17
product_space type ... 34
quantifier_expression entity ... 36
rationalize_function entity ... 90
real_interval type ... 35
real_interval_from_min entity ... 48
real_interval_to_max entity ... 49
real_max function ... 225
real_min function ... 225
real_tuple_literal entity ... 40
regular_indexing function ... 226
regular_table_function entity ... 76
reindexed_array_function entity ... 63
remove_first function ... 227
repackage function ... 227
repackage_options type ... 32
repackaging_function entity ... 61
restriction_function entity ... 60
schema_prefix constant ... 15
selector_function entity ... 59
series_composed_function entity ... 64
shape_of_array function ... 228

© ISO 2002 — All rights reserved

simplify_function_application function 229
simplify_generic_expression function 243
simplify_maths_space function 249
simplify_maths_value function 250
singleton_member_of function 251
space_constraint_type type 31
space_dimension function 252
space_is_continuum function 252
space_is_singleton function 253
standard_table_function entity 75
strict_triangular_matrix entity 78
stripped_typeof function 254
subspace_of function 254
subspace_of_es function 260
substitute function 261
symmetric_banded_matrix entity 80
symmetric_matrix entity 79
symmetry_type type 24
table function definition 5
the_binarys constant 15
the_booleans constant 15
the_complex_numbers constant 15
the_complex_tuples constant 16
the_empty_atom_based_tuple constant 16
the_empty_atom_based_value constant 16
the_empty_maths_tuple constant 16
the_empty_maths_value constant 16
the_empty_space constant 15
the_generics constant 15
the_integer_tuples constant 16
the_integers constant 15
the_logicals constant 15
the_maths_spaces constant 15
the_neg1_one_interval constant 15
the_neghalfpi_halfpi_interval constant 15
the_negpi_pi_interval constant 15
the_nonnegative_reals constant 15
the_numbers constant 15
the_real_tuples constant 16
the_reals constant 15
the_strings constant 15
the_tuples constant 16
the_zero_one_interval constant 15
the_zero_pi_interval constant 15
the_zero_tuple_space constant 16
triangular_matrix entity 78
tuple definition 5
tuple_space type 35
uniform_product_space entity 52
values_space_of function 263
variable definition 5
zero_or_one type 17

 © ISO 2002 — All rights reserved

ICS 25.040.40

Price based on 288 pages

© ISO 2002 – All rights reserved