APPENDIX A

Syntax Summary

Specifications

specification ::=
  module_decl-string

module_decl ::=
  scheme_decl |object_decl

Declarations

decl ::=
  scheme_decl |object_decl |
  type_decl |value_decl |
  variable_decl |channel_decl |
  axiom_decl

Scheme Declarations

scheme_decl ::=
  scheme scheme_def-list

scheme_def ::=
  opt-comment-string
  id opt-formal_scheme_parameter = class_expr

formal_scheme_parameter ::=
  (formal_scheme_argument-list)

formal_scheme_argument ::=
  object_def

Object Declarations

object_decl ::=
  object object_def-list

object_def ::=
  opt-comment-string
  id opt-formal_array_parameter : class_expr

formal_array_parameter ::=
  [typing-list]

Type Declarations

type_decl ::=
  type type_def-list

type_def ::=
  sort_def |
  variant_def |
  union_def |
  short_record_def |
  abbreviation_def

Sort Definitions

sort_def ::=
  opt-comment-string id

Variant Definitions

variant_def ::=
  opt-comment-string id == variant-choice

variant ::=
  constructor |
  record_variant

record_variant ::=
  constructor (component_kind-list)

component_kind ::=
  opt-destructor type_expr opt-reconstructor

constructor ::=
  id_or_op |
  —

destructor ::=
  id_or_op :

reconstructor ::=
  ↔ id_or_op

Union Definitions

union_def ::=
  opt-comment-string
  id = name_or_wildcard-cholce2

name_or_wildcard ::=
  type-name |
  —

Short Record Definitions

short_record _def ::=
  opt-comment-string
  id :: component_kind-string

Abbreviation Definitions

abbreviation_def ::=
  opt-comment-string id = type_expr

Value Declarations

value_decl ::=
  value value_def-list

value_def ::=
  commented_typing |
  explicit_value_def |
  implicit_value_def |
  explicit_function_def |
  implicit_function_def

Explicit Value Definitions

explicit_value_def ::=
  opt-comment-string
  single_typing = *pure*-value_expr

Implicit Value Definitions

implicit_value_def ::=
  opt-comment-string
  single_typing *pure*-restriction

Explicit Function Definitions

explicit_function_def ::=
  opt-comment-string single_typing
  formal_function_application ≡ value_expr
  opt-pre_condition

formal_function_application ::=
  id_application |
  prefix_application |
  infix_application

id_application ::=
  value-id formal_function_parameter-string

formal_function_parameter ::=
  (opt-binding-list)

prefix_application ::=
  prefix_op id

infix_application ::=
  id infix_op id

Implicit Function Definitions

implicit_function_def ::=
  opt-comment-string
  single_typing formal_function_application
  post_condition opt-pre_condition

Variable Declarations

variable_decl ::=
  variable variable_def-list

variable_def ::=
  single_variable_def |
  multiple_variable_def

single_variable_def ::=
  opt-comment-string
  id : type_expr opt-initialisation

initialisation ::=
  := *pure*-value_expr

multiple_variable_def ::=
  opt-comment-string id-list2 : type_expr

Channel Declarations

channel_decl ::=
  channel channel_def-list

channel_def ::=
  single_channel_def |
  multiple_channel_def

single_channel_def ::=
  opt-comment-string id : type_expr

multiple_channel_def ::=
  opt-comment-string id-list2 : type_expr

Axiom Declarations

axiom_decl ::=
  axiom opt-axiom_quantification
  axiom_def-list

axiom_quantification ::=
  forall typing-list •

axiom_def ::=
  opt-comment-string opt-axiom_naming
  readonly_logical-value_expr

axiom_naming ::=
  [id]

Class Expressions

class_expr ::=
  basic_class_expr |
  extending_class_expr |
  hiding_class_expr |
  renayming_class_expr |
  scheme_instantiation

Basic Class Expressions

basic_class_expr ::=
  class opt-decl-string **end**

Extending Class Expressions

extending_class_expr ::=
  extend class_expr **with** class_expr

Hiding Class Expressions

hiding_class_expr ::=
  hide defined_item-list **in** class_expr

Renaming Class Expressions

renaming_class_expr ::=
  use rename_pair-list **in** class_expr

Scheme Instantiations

scheme_instantiation ::=
  scheme-name opt-actual_scheme_parameter

actual_scheme_parameter ::=
  (object_expr-list)

Rename Pairs

rename_pair ::=
  defined_item **for** defined_item

Defined Items

defined_item ::=
  id_or_op |
  disambiguated_item

disambiguated_item ::=
  id_or_op : type_expr

Object Expressions

object_expr ::=
  object-name |
  element_object_expr |
  array_object_expr |
  fitting_object_expr

Element Object Expressions

element_object_expr ::=
  array-object_expr actual_array_parameter

actual_array_parameter :: =
  [pure-value_expr-list]

Array Object Expressions

array_object_expr ::=
  [| typing-list • *element*-object_expr |]

Fitting Object Expressions

fitting_object_expr ::=
  object_expr { rename_pair-list }

Type Expressions

type_expr ::=
  type_literal |
  type-name |
  product_type_expr |
  set_type_expr |
  list_type_expr |
  map_type_expr |
  function_type_expr |
  subtype_expr |
  bracketed_type_expr

Type Literals

type_literal ::=
  Unit |**Bool** |**Int** |**Nat** |**Real** |**Text** |
  Char

Product Type Expressions

product_type_expr ::=
  type_expr-product2

Set Type Expressions

set_type_expr ::=
  finite_set_type_expr |
  infinite_set_type_expr

finite_set_type_expr ::=
  type_expr-set

infinite_set_type_expr :: =
  type_expr-infset

List Type Expressions

list_type_expr ::=
  finite_list_type_expr |
  infinite_list_type_expr

finite_list_type_expr ::=
  type_expr*

infinite_list_type_expr ::=
  type_expr*

Map Type Expressions

map_type_expr ::=
  type_expr * type_expr

Function Type Expressions

function_type_expr ::=
  type_expr function_arrow result_desc

function_arrow ::=
  → |∏ |≅

result_desc ::=
  opt-access_desc-string type_expr

Subtype Expressions

subtype_expr ::=
  {| single_typing *pure*-restriction |}

Bracketed Type Expressions

bracketed_type_expr ::=
  (type_expr)

Access Descriptions

access_desc ::=
  access_mode access-list

access_mode ::=
  read |**write** |**in** |**out**

access ::=
  variable_or_channel-name |
  enumerated_access |
  completed_access |
  comprehended_access

enumerated_access ::=
  { opt-access-list }

completed_access ::=
  opt-qualification any

comprehended_access ::=
  { access | pure-set_limitation }

Value Expressions

value_expr ::=
  value_literal |
  value_or_variable-name |
  pre_name |
  basic_expr |
  product_expr |
  set_expr |
  list_expr |
  map_expr |
  function _expr |
  application_expr |
  quantified_expr |
  equivalence_expr |
  post_expr |
  disambiguation_expr |
  bracketed_expr |
  infix_expr |
  prefix_expr |
  comprehended_expr |
  initialise_expr |
  assign ment_expr |
  input_expr |
  output_expr |
  structured_expr

Value Literals

value_literal ::=
  unit_literal |
  bool_literal |
  int_literal |
  real_literal |
  text_literal |
  char_literal

unit_literal ::=
  ( )

bool_literal ::=
  true |**false**

Pre Names

pre_name ::=
  variable-name

Basic Expressions

basic_expr ::=
  chaos |skip |stop |swap

Product Expressions

product_expr ::=
  (value_expr-list2)

Set Expressions

set_expr ::=
  ranged_set_expr |
  enumerated_set_expr |
  comprehended_set_expr

Ranged Set Expressions

ranged_set_expr ::=
  { *readonly_integer*-value_expr ..
    readonly_integer-value_expr }

Enumerated Set Expressions

enumerated_set_expr ::=
  { *readonly*-opt-value_expr-list }

Comprehended Set Expressions

comprehended_set_expr ::=
  { readonly-value_expr | set_limitation }

set_limitation ::=
  typing-list opt-restriction

restriction ::=
  • *readonly_logical*-value_expr

List Expressions

list_expr ::=
  ranged_list_expr |
  enumerated_list_expr |
  comprehended_list_expr

Ranged List Expressions

ranged_list_expr ::=
  ⟨ *integer*-value_expr .. *integer*-value_expr ⟩

Enumerated List Expressions

enumerated_list_expr ::=
  ⟨ opt-value_expr-list ⟩

Comprehended List Expressions

comprehended_list_expr ::=
  ⟨ value_expr | list_limitation ⟩

list_limitation ::=
  binding **in** *readonly_list*-value_expr
  opt-restriction

Map Expressions

map_expr ::=
  enumerated_map_expr |
  comprehended_map_expr

Enumerated Map Expressions

enumerated_map_expr ::=
  [opt-value_expr_pair-list]

value_expr_pair ::=
  readonly-value_expr |→ *readonly*-value_expr

Comprehended Map Expressions

comprehended_map_expr ::=
  [value_expr_pair | set_limitation]

Function Expressions

function_expr ::=
  λ lambda_parameter • value_expr

lambda_parameter ::=
  lambda_typing |
  single_typing

lambda_typing ::=
  (opt-typing-list)

Application Expressions

application_expr ::=
  list_or_map_or_function-value_expr
actual_function_parameter-string
  actual_function_parameter (opt-value_expr-list)

Quantified Expressions

quantified_expr ::=
  quantifier typing-list restriction

quantifier ::=
  ∀ |∃ |∃!

Equivalence Expressions

equivalence_expr ::=
  value_expr ≡ value_expr opt-pre_condition

pre_condition ::=
  pre *readonly*_logical-value_expr

Post Expressions

post_expr ::=
  value_expr post_condition opt-pre_condition

post_condition ::=
  opt-result_naming **post** *readonly_logical*-value_expr

result_naming ::=
  as binding

Disambiguation Expressions

disambiguation_expr ::=
  value_expr : type_expr

Bracketed Expressions

bracketed_expr ::=
  (value_expr)

Infix Expressions

infix_expr ::=
  stmt_infix_expr|
  axiom_infix_expr |
  value_infix_expr

Statement Infix Expressions

stmt_infix_expr ::=
  value_expr infix_combinator value_expr

Axiom Infix Expressions

axiom_infix_expr ::=
  logical-value-expr infix-connective
  logical-value_expr

Value Infix Expressions

value_infix_expr ::=
  value_expr infix_op value_expr

Prefix Expressions

prefix_expr ::=
  axiom_prefix_expr |
  universal_prefix_expr |
  value_prefix_expr

Axiom Prefix Expressions

axiom_prefix_expr ::=
  prefix_connective *logical*-value_expr

Universal Prefix Expressions

universal_prefix_expr ::=
  □ *readonly_logical*-value_expr

Value Prefix Expressions

value_prefix_expr ::=
  prefix_op value_expr

Comprehended Expressions

comprehended_expr ::=
  associative_commutative-infix_combinator
  { value_expr | set_limitation }

Initialise Expressions

initialise_expr ::=
  opt-qualification **initialise**

Assignment Expressions

assignment_expr ::=
  variable-name := value_expr

Input Expressions

input_expr ::=
  channel-name ?

Output Expressions

output_expr ::=
  channel-name ! value_expr

Structured Expressions

structured_expr ::=
  local_expr |let_expr|
  if_expr |case_expr |
  while_expr |until_expr |
  for_expr

Local Expressions

local_expr ::=
  local opt-decl-string **in** value_expr **end**

Let Expressions

let_expr ::=
  let let_def-list **in** value_expr **end**

let_def ::=
  typing |
  explicit_let |
  implicit_let

explicit_let ::=
  let_binding = value_expr

implicit_let ::=
  single_typing restriction

let_binding ::=
  binding |
  record_pattern |
  list_pattern

If Expressions

if_expr ::=
  if *logical*-value_expr **then**
  value_expr
  opt-elsif_branch-string
  opt-else_branch
  end

elsif_branch ::=
  elsif *logical*-value_expr **then** value_expr

else_branch ::=
  else value_expr

Case Expressions

case_expr ::=
  case value_expr **of** case_branch-list **end**

case_branch ::=
  pattern → value_expr

While Expressions

while_expr ::=
  while *logical*-value_expr
  do unit-value_expr **end**

Until Expressions

until_expr ::=
  do *unit*-value_expr
  until *logical*-value_expr **end**

For Expressions

for_expr ::=
  for list_limitation **do** *unit*-value_expr **end**

Bindings

binding ::=
  id_or_op |
  product_binding

product_binding ::=
  (binding-list2)

Typings

typing ::=
  single_typing |
  multiple_typing

single_typing ::=
  binding : type_expr

multiple_typing ::=
  binding-list2 : type_expr

commented_typing ::=
  opt-comment-string typing

Patterns

pattern ::=
  value_literal |
  pure_value-name |
  wildcard_pattern |
  product_pattern |
  record_pattern |
  list_pattern

Wildcard Patterns

wildcard_pattern ::=
  —

Product Patterns

product_pattern ::=
  (inner_pattern-llst2)

Record Patterns

record_pattern ::=
  pure_value-name (inner_pattern-list)

List Patterns

list_pattern ::=
  enumerated_list_pattern |
  concatenated_list_pattern

Enumerated List Patterns

enumerated_list_pattern ::=
  (opt-inner_pattern-llst)

Concatenated List Patterns

concatenated_list_pattern ::=
  enumerated_list_pattern ^ inner_pattern

Inner Patterns

Inner_pattern ::=
  value_literal |
  id_or_op |
  wildcard_pattern |
  product_pattern |
  record_pattern |
  list_pattern |
  equality_pattern

Equality Patterns

equality_pattern ::=
  = *pure_value*-name

Names

name ::=
  qualified_id |qualified_op

Qualified Identifiers

qualified_id ::=
  opt-qualification id

qualification ::=
  element-object_expr

Qualified Operators

qualified_op ::=
  opt-qualification (op)

Identifiers and Operators

id_or_op ::=
  id |op

op ::=
  infix_op |prefix_op

Infix Operators

infix_op ::=
  = | ≠ | > |< |≥ |≤ |⊃ |⊂ |⊇ |⊆ |∈ |∉ |+ |– |\ |^ |∪ |
  † |* | / |° |∩ |↑

Prefix Operators

prefix_op ::=
  abs |**int** |**real** |**card** |**len** |**inds** |
  elems |**hd** |**tl** |**dom** |**rng**

Connectives

connective ::=
  infix_connective |
  prefix_connective

Infix Connectives

infix_connective ::=
  ⇒ |∨ |∧

Prefix Connectives

prefix_connective ::=
  ~

Infix Combinators

infix_combinator ::=
  □ |□ |□ |□ |;

APPENDIX B

Precedence and Associativity of Operators

Value operator precedence - increasing		
Prec	Operator(s)	Assotiativity
14	□ λ ∀ ∃ ∃!	Right
13	≡ **post**	
12	□ ∏ □□	Right
11	;	Right
10	:=	
9	⇒	Right
8	∨	Right
7	∧	Right
6	= ≠ > < ≥ ≤ ⊂ ⊆ ⊃ ⊇ ∈ ∉	
5	+ − □ □ ∪ †	Left
4	* / ° ∩	Left
3	↑	
2	:	
1	~ prefix_op	

Type operator precedence - increasing		
Prec	Operator(s)	Associativity
3	→	Right
2	×	
1	**-set -infset** * **	

APPENDIX C

Lexical **Matters**

This chapter describes lexical matters, i.e. the microsyntax for RSL.

Basically, RSL follows the rules now in current practice for most programming languages:text (i.e. an RSL specification) is represented as a string of characters, which is interpreted left-to-right and broken into a string of tokens. The characters are drawn from a superset of the ASCIIcharacters called the *full RSL character set.* Tokens may be separated by 'whitespace', which isstrings of one or more of the following characters: line-feed, carriage-return, space and tab. (Note that *comments* are part of the RSL syntax and thus cannot be used freely as whitespace. Also note that comments may not be nested.)

There are two types of tokens in RSL: varying and fixed.

Varying Tokens

The microsyntax for varying tokens is defined by the syntax rules below, where the charactersused in forming tokens are shown in quotes, as in '$'. Furthermore, LF, CR and TAB are used to denote the ASCII characters line-feed, carriage-return and tab.

id ::=
  letter opt-letter_or_digit_or_underline_or_prime-string

letter_or_digit_or_underline_or_prime ::=
  letter | digit | underline | prime

letter ::=

ascii_letter | greek_letter

comment ::=
‘/’ ‘*’ comment_item-string ‘*’ ’/’

comment_item ::=
comment_char

comment_char ::=
LF | CR | TAB | ascii_letter | digit | graphic | prime | quote | backslash

int_literal ::=
digit-string

real_literal ::=
digit-string ’.’ digit-string

text_literal ::=
‘. .’opt-text_character-string ‘. .’

char_literal ::=
‘.’char_character ‘.’

text_character ::=
character | prime

char_character ::=
character | quote

character ::=
ascii_letter | digit | graphic | escape

digit ::=
‘0’ |’1’ |’ 2’ |’ 3’ |’ 4’ |’ 5’ |’ 6’ |’ 7’ |’ 8’ |’ 9’

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

greek_letter ::=
‘`alpha’|’`beta’|’`gamma’|’`delta’|’`epsilon’|’`zeta’|’`eta’|’`theta’|’`iota’|‘`kappa’|’`mu’|’`nu’|’`xi’|’`pi’|’`rho’|’`sigma’|
’`tau’|’`upsilon’|’`phi’|’`chi’|’`psi’|’`omega’|’`Gamma’|’`Delta’|’`Theta’|’`Lambda’|’`Xi’|’`Pi’|’`Sigma’|’`Upsilon’|
’`Phi’|’`Psi’|’`Omega’

underline ::=
‘_’

prime ::=
‘.’

quote ::=
‘. .’

backslash ::=
‘\’

graphic ::=
‘ ’ | ‘!’ | ‘#’ | ‘$’ | ‘%’ | ‘&’ | ‘(’ | ‘)’ | ‘*’ | ‘+’ | ‘,’ | ‘-’ | ‘.’ | ‘/’ | ‘:’ | ‘;’ |‘<’ | ‘=’ | ‘>’ | ‘?’ | ‘@’ | ‘[’ | ‘]’ | ‘^’ | ‘_’ | ‘`’ |
‘{’ | ‘|’ | ‘}’ | ‘~’

escape ::=
‘\’’r’|‘\’’n’|‘\’’t’|‘\’’a’|‘\’’b’|‘\’’f’|‘\’’v’|‘\’’?’|‘\’’\’|‘\’’ .’|‘\’’ . .’| ‘\’ oct_constant | ‘\’ ‘x’ hex_constant

oct_constant ::= oct_digit | oct_digit oct_digit | oct_digit oct_digit oct_digit

hex_constant ::=
hex_digit-string

oct_digit ::=
‘0’ |’1’ |’ 2’ |’ 3’ |’ 4’ |’ 5’ |’ 6’ |’ 7’

hex_digit ::=
digit | ‘a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ‘A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’

ASCII forms of Greek Letters

Greek letters, which may be used in identifiers, have ASCII forms are follows :

ASCII	Full	ASCII	Full
`alpha	α		
`beta	β		
`gamma	γ	`Gamma	Γ
`delta	δ	`Delta	Δ
`epsilon	ε		
`zeta	ζ		
`eta	η		
`theta	θ	`Theta	Θ
`iota	ι		
`kappa	κ		
		`Lambda	Λ
`mu	μ		
`nu	ν		
`xi	ξ	`Xi	Ξ
`pi	π	`Pi	Π
`rho	ρ		
`sigma	σ	`Sigma	Σ
`tau	τ		
`upsilon	υ	`Upsilon	Υ
`phi	ϕ	`Phi	Φ
`chi	χ		
`psi	φ	`Psi	Ψ
`omega	ω	`Omega’	Ω

Fixed Tokens

The representation of individual fixed token is given directly in the syntax rules for RSL. However, a representation using only ASCII characters is possible, as denied in the following table:

ASCII	Full	ASCII	Full	ASCII	Full
><	×	isin	∈	~isin	∉
||	□	++	□	-\	λ
|=|	□	|^|	∏	-list	*
**	↑	- inflist		~=	≠
/\	∧	\/	∨	+>	↦
>=	≥	exists	∃	all	∀
<=	≤	union	∪	!!	†
inter	∩	<<	⊂	always	□
-m->		<<=	⊆	=>	⇒
-~->		>>	⊃	is	≡
->	→	>>=	⊇	<->	↔
#	°	<.	⟨	.>	⟩
:-	•				

The word equivalents of certain symbols: `all`, `exists`, `union`, `inter`, `isin`, `always` are reserved, and cannot be used as identifiers.

RSL Keywords

The RSL keywords are listed below. They cannot be used as identifiers.

Keywords for RSL			
Bool	**class**	**inds**	**skip**

Char	**do**	**initialise**	**stop**
Int	**dom**	**int**	**swap**
Nat	**elems**	**len**	**then**
Real	**else**	**let**	**tl**
Text	**elsif**	**local**	**true**
Unit	**end**	**object**	**type**
abs	**extend**	**of**	**until**
any	**false**	**out**	**use**
as	**for**	**post**	**value**
axiom	**forall**	**pre**	**variable**
card	**hd**	**read**	**while**
case	**hide**	**real**	**with**
channel	**if**	**rng**	**write**
chaos	**in**	**scheme**	