An RSL Subset (V1)

Anne E. Haxthausen

DTU Compute, February 14, 2024

Contents

CHAPTER 1

Syntax for RSL Subset V1

This section presents a grammar for a subset of RSL. The subset excludes the following constructs from full RSL:

1. module structuring constructs, i.e.:
   (a) object delarations are excluded,
   (b) a specification now consists of only *one* scheme definition,
   (c) the scheme definition must not have formal parameters,
   (d) its constituent class expression must be a basic class expression - all other kinds of class expressions are disallowed, and
   (e) a basic class expression can now only contain type, value, and axiom declarations (not variable, channel, and object declarations)
2. multiple typings (only single typings are allowed)
3. product bindings (so only id is allowed as a binding)
4. implicit value definitions
5. implicit function definitions (using pre/post specs)
6. user-defined operators
7. imperative constructs (e.g. variable declarations, assignments and loops)
8. process/concurrency constructs, (e.g. channel declarations, input and output expressions, and concurrency combinators)
9. types and associated value expressions and operators for natural numbers, reals, texts (strings), chars, the unit value, products (tuples), functions, sets, lists, maps (tables), union, records, and record variants (only variants corresponding to enumerations are allowed)
10. case expressions
11. elsif branches inside if-then-else constructs
12. local expressions
13. equivalence expressions
14. post expressions
15. higher-order functions and higher-order function application expressions, e.g. application_expr can only be on the form id(value_expr-list)

Changes to the grammar

The grammar for the subset of RSL presented in this chapter has been achieved from the grammar of the full RSL language by removing many alternatives and sometimes making optional constructs either non-optional or totally removing them. Furthermore, in some cases, in the right-hand side of a rule *t ::= ... t1 ...*, a non-terminal *t1* has been replaced with the right-hand side *rhs* of the grammar rule *t1 ::= rhs* for *t1* (when the right-hand side *rhs* only had one alternative) and the rule for *t1* was removed. So the first rule became *t ::= ... rhs ....*

As an example, the original grammar of bindings was:

binding ::=
  id |
  id product_binding

product_binding ::=
  (binding-list2)

Then first *product_binding* was removed from the language, resulting in the grammar:

binding ::=
  id

Then all appearences of *binding* in right-hand sides of grammar rules were replaced with *id* and the rule for *binding* was removed.

The first kind of grammar changes was necessary for reducing to the sub-language. Later on, it would be easy to re-introduce the removed constructs, e.g. by adding an alternative. The second kind of grammar changes were not necessary, but only made in order to simplify the grammar and make parse trees smaller. A disadvantage of the latter kind of grammar changes is that it becomes less easy to re-introduce removed constructs. However, it was judged that reducing the size of parse trees was more important as we had examples where large parse trees were an issue.

Suggested changes to static semantics

The static semantics is restricted by:

1. disallowing overloading
2. disallowing recursive functions
3. maybe enforcing the define-before-use paradigm and later remove this restriction???

Scheme Declarations

scheme_decl ::=
  scheme scheme_def

scheme_def ::=
  id = class_expr

Class Expressions

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

Declarations

decl ::=
  type_decl |
  value_decl |
  axiom_decl

Type Declarations

type_decl ::=
  type type_def-list

type_def ::=
  sort_def |
  variant_def |
  abbreviation_def

sort_def ::=
  id

variant_def ::=
  id == id-choice

abbreviation_def ::=
  id = type_expr

Value Declarations

value_decl ::=
  value value_def-list

value_def ::=
  value_signature |
  explicit_value_def |
  explicit_function_def

value_signature ::=
  id : type_expr

explicit_value_def ::=
  id : type_expr = *pure*-value_expr

explicit_function_def ::=
  id : type_expr-product $\xrightarrow{\sim}$ type_expr
  id (id-list) ≡ value_expr

Variable Declarations

variable_decl ::=
  variable variable_def-list

variable_def ::=
  id : type_expr opt-initialisation

initialisation ::=
  := *pure*-value_expr

NOTE non-terminal variable_decl is not used in any grammar rule for the defined RSL subset, but in the grammar rules for the RSL* extension. Since variable_decl stems from full RSL, it is placed here.

Axiom Declarations

axiom_decl ::=
  axiom axiom_def-list

axiom_def ::=
  readonly_logical-value_expr

axiom_naming ::=
  [id]

Type Expressions

type_expr ::=
  type_literal |
  type-name |
  subtype_expr |
  bracketed_type_expr

type_literal ::=
  Bool |
  Int

subtype_expr ::=
  {| single_typing • *pure_logical*-value_expr |}

bracketed_type_expr ::=
(type_expr)

Value Expressions

value_expr ::=
value_literal |
value_or_variable-name |
application_expr |
quantified_expr |
bracketed_expr |
axiom_infix_expr |
value_infix_expr |
axiom_prefix_expr |
value_prefix_expr |
let_expr |
if_expr

value_literal ::=
bool_literal |
int_literal

bool_literal ::=
true |
false

application_expr ::=
function_value-id (value_expr-list)

quantified_expr ::=
quantifier single_typing-list •
pure_logical-value_expr

quantifier ::=
∀ |
∃ |
∃!

bracketed_expr ::=
(value_expr)

axiom_infix_expr ::=
logical-value_expr infix_connective
logical-value_expr

value_infix_expr ::=
value_expr infix_op value_expr

axiom_prefix_expr ::=
prefix_connective *logical*-value_expr

value_prefix_expr ::=
prefix_op value_expr

let_expr ::=
let id = value_expr **in** value_expr **end**

if_expr ::=
if *logical*-value_expr
then value_expr
else value_expr
end

Typings

single_typing ::=
id : type_expr

Names

name ::=
id

Operators and Connectives

infix_op ::=
= |
≠ |
> |
< |
≥ |
≤ |
+ |
− |
* |
/

prefix_op ::=
abs

infix_connective ::=
⇒ |
∨ |
∧

prefix_connective ::=
∼

Precedence and Associativity of Operators for Full RSL

This section is taken from the RSL book.

Value operator precedence – increasing		
Prec	Operator(s)	Associativity
14	$\Box\ \lambda\ \forall\ \exists\ \exists!$	Right
13	$\equiv$ **post**	
12	$[]\ \sqcap\ \|\|\ \|\!\|$	Right
11	;	Right
10	:=	
9	$\Rightarrow$	Right
8	$\vee$	Right
7	$\wedge$	Right
6	$= \neq > < \geq \leq \subset \subseteq \supset \supseteq \in \notin$	
5	$+\ -\ \setminus\ \frown\ \cup\ \dagger$	Left
4	$*\ /\ \circ\ \cap$	Left
3	$\uparrow$	
2	:	
1	$\sim$	

Type operator precedence – increasing		
Prec	Operator(s)	Associativity
3	$\overrightarrow{m}\ \xrightarrow{\sim}\ \rightarrow$	Right
2	$\times$	
1	**-set -infset** $^*\ ^\omega$	

CHAPTER 3

Lexical Matters for Full RSL

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

Basically, RSL follows the rules now in current practice for most programming languages: a 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 ASCII characters called the *full RSL character set*. Tokens may be separated by 'whitespace', which is strings 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 characters used 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 as 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	ψ	`Psi	Ψ
`omega	ω	`Omega	Ω

Fixed Tokens

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

ASCII	Full	ASCII	Full	ASCII	Full
`><`	$\times$	`isin`	$\in$	`~isin`	$\notin$
`||`	$\parallel$	`++`	$+\!\!+$	`-\`	λ
`|=|`	$\square$	`|^|`	$\coprod$	`-list`	*
`**`	$\uparrow$	`-inflist`	$^\omega$	`~=`	$\neq$
`/\`	$\wedge$	`\/`	$\vee$	`+>`	$\mapsto$
`>=`	$\geq$	`exists`	$\exists$	`all`	$\forall$
`<=`	$\leq$	`union`	$\cup$	`!!`	$\dagger$
`inter`	$\cap$	`<<`	$\subset$	`always`	$\Box$
`-m->`	$\overrightarrow{m}$	`<<=`	$\subseteq$	`=>`	$\Rightarrow$
`-~->`	$\xrightarrow{\sim}$	`>>`	$\supset$	`is`	$\equiv$
`->`	$\rightarrow$	`>>=`	$\supseteq$	`<->`	$\leftrightarrow$
`#`	$\circ$	`<.`	$\langle$	`.>`	$\rangle$
`:-`	$\bullet$				

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**	

About Syntax and Context Rules

This chapter is an extract from the RSL text book. Therefore, some of the mentioned syntactic categories do not exist in the grammar for the RSL subset presented in the present document.

Syntax Conventions

The syntax contains one or more syntax rules each of the form:

category_name ::=
  alternative$_1$|
  ...
  alternative$_n$

where $\mathbf{n} \geq 1$. This rule introduces syntax category (non-terminal) named category_name and defines that category as the union of the strings generated by the alternatives. As an example consider:

set_type_expr ::=
  finite_set_type_expr|
  infinite_set_type_expr

Each alternative consists of a sequence of tokens where a token is of one of three kinds:

- A keyword in bold font such as '**Bool**'
- A symbol such as '('.
- A subcategory name such as 'value_expr', possibly prefixed with a text such as '*logical-*' in italics.

The strings generated by an alternative are those obtained by concatenating keywords, symbols and strings from subcategories — in the order of appearance. As examples consider:

finite_set_type_expr ::=
  type_expr**-set**

map_type_expr ::=
  type_expr $\overrightarrow{m}$ type_expr

The convention below is used for defining optional presence ('nil-x' represents absence of 'x'): For any syntax category name 'x' the following rule is assumed:

opt-x ::=
  nil-x|
  x

The conventions below are used for defining repetition: For any syntax category name ‘x’ the following rules are assumed:

x-string ::=
  x|
  x x-string

x-list ::=
  x|
  x , x-list

x-list2 ::=
  x , x|
  x , x-list2

x-choice ::=
  x|
  x | x-choice

x-choice2 ::=
  x | x|
  x | x-choice2

Note that ‘|’ is an RSL symbol in ‘x | x’ and ‘x | x-choice2’.

x-product2 ::=
  x × x|
  x × x-product2

x-product ::=
  x|
  x × x-product

If ‘opt’ occurs together with ‘string’ or ‘list’, ‘opt’ has the lower precedence. That is, for any syntax category name ‘x’ the following rules are assumed:

  opt-x-string ::=
    nil-x-string|
    x-string

  opt-x-list ::=
    nil-x-list|
    x-list

Similarly, if ‘nil’ occurs together with ‘string’ or ‘list’, ‘nil’ has the lower precedence.

The conventions below are used for indicating context conditions:

If a category name appearing in an alternative is prefixed with a word in italics, then this word indicates a context condition, as explained in the tables A.1–A.3. As an example consider the following syntax rule, where the context condition is that the maximal type of the constituent value expression must be **Bool**:

  axiom_prefix_expr ::=

prefix_connective *logical*-value_expr

If a category name appearing in an alternative is prefixed with several words in italics separated by underscores, then each of the words indicates a context condition. As an example consider the following syntax rule, where the context conditions are that the constituent value expression must be read-only and have the maximal type **Bool**:

restriction ::=
- *readonly_logical*-value_expr

If a category name appearing in an alternative is prefixed with a text containing several words in italics separated by '*_or_*', then this text indicates a context condition which is the disjunction of each of the individual context conditions (i.e. one of the context conditions must be fulfilled). As an example consider the following syntax rule, where the context condition is that the constituent name must represent a value or a variable:

value_expr ::=
value_or_variable-name

prefix	context condition
unit	the maximal type of the value_expr must be **Unit**
logical	the maximal type of the value_expr must be **Bool**
integer	the maximal type of the value_expr must be **Int**
list	the maximal type of the value_expr must be a list type
map	the maximal type of the value_expr must be a map type
function	the maximal type of the value_expr must be a function type
pure	the value_expr must be pure
readonly	the value_expr must be read-only

Table A.1: Prefixes of value_expr and the context conditions they indicate

prefix	context condition
pure	the maximal type of the name must be a pure function type
type	the name must represent a type
value	the name must represent a value
variable	the name must represent a variable
channel	the name must represent a channel
scheme	the name must represent a scheme
object	the name must represent an object

Table A.2: Prefixes of name and the context conditions they indicate

Static Correctness

This section presents an overview of the common context conditions of RSL.

prefix	context condition
value	the id must represent a value

Table A.3: Prefixes of id and the context conditions they indicate

A syntactically correct string is *statically correct* if its context conditions hold. The description *static* indicates that static correctness can be checked decidably (i.e. by a terminating mechanical process). This means that we need to distinguish between what is statically true and what might be proved to be actually true. In particular we need to distinguish between static types (which we call 'maximal') and actual types of expressions, and between static accesses (the variables and channels an expression can access) and the actual accesses.

The main context conditions are:

1. All definitions having the same scope must be compatible.
2. All applied occurrences of operators and identifiers must be within the scopes of their definitions and these definitions must be visible.
3. It must be possible to associate uniquely and consistently a 'maximal' type with each operator and identifier representing a value, variable or channel. (Note: in RSL* V1, user defined operators are not allowed.)
4. The accesses to variables and channels of the bodies of explicit function definitions must be allowed by the access descriptions in their signatures. (Note: accesses are not part of RSL* V1.)

1. The first condition about compatible definitions prevents things like:

type
T = **Int**,
T = **Bool**

In general different identifiers must be used for different things. There are some exceptions to this — see book chapter on overloading. In RSL* subset 1, we do not allow overloading.

2. The second condition about definitions being visible is similar to the standard rule in block structured programming languages. But note that RSL does not have any 'define before use' rule. More details about scope and visibility can be found in RSL book chapter 34.

3. The third condition about type consistency bans expressions like '1 + **true**'.

The rule is concerned with 'maximal' types since, for example, it is statically undecidable whether a particular integer expression will evaluate to give a natural number. So it is not against the context conditions to, for example, divide by zero or to take the head of an empty list, but the meaning of such an expression is

typically under-specified in the semantics of RSL (which means that the specifier may not predict how the final implementation will behave).

4. The fourth condition ensures that the actual accesses made to variables and channels in the bodies of functions correspond to the accesses allowed in their signatures. So a function is only allowed to read a variable if it has read (or write) access to it, only allowed to write to a variable if it has write access to it, only allowed to input from a channel if it has input access to it and only allowed to output to a channel if it has output access to it.