RSL Reference Manual

Part No.: RAISE/CRI/DOC/2/V1

Date: April 6, 1990

Original Authors: Klaus Havelund,
Anne Haxthausen

Note

This document is a pre-release produced for the VDM '90 RAISE tutorial.

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to describe the RAISE Specification Language, RSL. The description is supposed to be suited for ‘looking up’ information rather than for ‘sequential reading’. It is a manual rather than a tutorial.

1.2 Target group

The target group of this document is users of RSL.

1.3 Relations to other documents

A prerequisite for reading this document is familarity with the RSL tutorial [1].

1.4 Structure of document

The document is formally structured over the syntax of RSL (see below). The introduction is followed by a special section on declarative constructs and a special section on overloading, and after that a section on each of the main syntax categories of RSL:

- Declarative constructs and visibility rules
- Overloading
- Specifications
- Declarations
- Type expressions
- Expressions
- Bindings
- Typings
- Patterns
- Names
- Infix combinators

- Connectives

Finally, the document contains a list of literature references, an index and three appendices. The first appendix describes lexical matters for RSL, the second apendix desribes precedence and associativity of RSL operators and the third appendix contains an RSL syntax summary.

1.5 Document conventions

The language description is centered around the syntax for RSL. The syntax defines the syntactically correct strings of the language. The strings are divided into syntax categories with the top syntax category containing all syntactically correct RSL specifications. Each syntax category is defined by a rule. The rules of the syntax are grouped into sections in the manual. Each section consists of some or all of the following subsections:

Syntax

Terminology

Meaning

Context conditions

Properties

Below the contents of these subsections is described and the used conventions are explained.

Syntax Contains one or more syntax rules each of the form

```
category_name ::=
  alternative_1 |
  ...
  alternative_n
```

where $n \geq 1$. This rule introduces the syntax category 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 bolded font such as '**Bool**'
- A symbol such as '('.
- A sub-category name such as '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 sub-categories – 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 below convention is used for defining optional presence (ϵ represents absence): For any syntax category name 'x' the following rule is assumed.

opt_x ::=
  ϵ|
  x

The below conventions 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-list

x−choice ::=
  x|
  x | x−choice

x−choice2 ::=
  x | x−choice

x−product2 ::=
  x × x−product

x−product ::=
  x|
  x × x−product

The below conventions are used for indicating context conditions:

If a category name appearing in an alternative is prefixed with a word in italics, then this word conveys a context condition, as explained in the tables 1 - 7. As an example consider the following syntax rule, where the conveyed context condition is that the maximal type of the constituent expression must be **Bool**:

axiom_prefix_expr ::=
  • *logical*-expr

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

restriction ::=
  • *readonly_logical*-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 convey a context condition which is the disjunction of each of the context conditions conveyed by the individual words (i.e. one of the context conditions conveyed by the individual words must be fulfilled). As an example consider the following syntax rule, where the conveyed context condition is that the constituent name must represent a value or a variable:

expr ::=
  value_or_variable-name

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

Table 1: Prefixes of expr and the context conditions they convey

Terminology Contains definitions of terms etc. When a term is defined it is written in italics.

Meaning Contains a description of the meaning of statically correct strings.

Context conditions Contains a description of the conditions that syntactically correct strings must satisfy in order to be statically correct. Note, that as a convenience some of these conditions are also indicated by italicized prefixes in the syntax rules, as described above.

Properties Contains a description of the properties that statically correct strings have. This information is used to describe context conditions.

prefix	context condition
pure	the restriction must be pure

Table 2: Prefixes of restriction and the context conditions they convey

prefix	context condition
pure	the set_limitation must be pure

Table 3: Prefixes of set_limitation and the context conditions they convey

prefix	context condition
pure	the name must be pure
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 4: Prefixes of name and the context conditions they convey

prefix	context condition
value	the id must represent a value

Table 5: Prefixes of id and the context conditions they convey

prefix	context condition
element	the object_expr must represent a model
array	the object_expr must represent an array

Table 6: Prefixes of object_expr and the context conditions they convey

prefix	context condition
associative	the infix_combinator must be associative
commutative	the infix_combinator must be commutative

Table 7: Prefixes of infix_combinator and the context conditions they convey

2 Declarative constructs and visibility rules

A *declarative construct* is a language construct representing one or more definitions. A *definition* introduces an identifier or operator for an entity such as a scheme, an object, a type, a value, a variable, a channel or an axiom. A definition stems from one of the following declarative constructs:

module_decl, decl, formal_scheme_parameter, formal_array_parameter, lambda_parameter, single_typing, typing, axiom_quantification, let_def, class_expr, object_expr, list_limitation, formal_function_application, result_naming, pattern

Notice, that some declarative constructs give first rise to definitions when they are in a context. For instance, a pattern gives first rise to definitions when a value in the context is matched against it. For such constructs the maximal types of the identifiers and/or operators introduced by the definitions is determined by a maximal type given by the context. Such a maximal type is called a *maximal context type* for (or of) the declarative construct.

A definition has an associated region of RSL text, called the *scope* of the definition. Within this scope, and only there, there are places where its entity may be referred to by its identifier or operator. We will talk also about the scope of a declarative construct meaning the scope of its definitions. The *scope rules* of the language determine the scope of definitions.

A definition is said to be *visible* at a point of RSL text if its entity may be referred to by its identifier or operator at that point. At such a point the identifier or operator is said to *represent* the entity or to be *a name of* the entity. The *visibility rules* of the language determine the visibility of definitions.

Two definitions are said to be *compatible* if they introduce distinct identifiers and operators or if they are both value definitions introducing the same identifier or operator but with distinguishable maximal types. Two declarative constructs are said to be compatible if all their definitions are compatible.

The context conditions ensure that at each point of RSL text all visible definitions are compatible.

Scope rules

The scope of a declarative construct depends on the context in which it occurs. Therefore for each construct containing a declarative construct the scope of this must be given. This is done in the subsections called "Properties" using the following conventions:

1. For declarative constructs occuring in non declarative constructs the scope is always explicitly stated. (This is for instance the case for the declarations in a local expression, see the example below.)

Example 2.1

```
local
  value
    x : Int = 3
in
  x + 2
end
```

The scope of the definition of x is the expression x + 2.
□

2. For declarative constructs occuring in declarative constructs there are the following possibilities:

   (a) The scope is explicitly stated. (This is for instance the case for the typings in an object definition, see the example below.)

   Example 2.2

   ```
   object
     O[i : Int] :
       class
         variable
           v : Int := i − 7
       end
   ```

   The scope of the definition of i is the class expression.
   □

   (b) An immediate scope is stated. (This is for instance the case for the declarations in a basic class expression, see the example below.) In this case the scope is the immediate scope plus possible extensions. The extensions depend on the context for the outer declarative construct and is given for all occurrences of it. (For instance for the class expressions in an extending class expression.)

   Example 2.3

   ```
   scheme
     S = extend
           class
             value
               x : Int = 3,
           end
         with
           value
             y : Int = x
         end
   ```

   The immediate scope of the definition of x is the region between **class** and **end**. The total scope of x is this region plus the region between **with** and **end**.
   □

(c) No scope is given. (This is for instance the case for the value definitions in a value declaration, see the example below.) In this case it is implicitly understood that the scope of the inner construct is given by the scope of the outer construct in which it occurs.

Example 2.4

value
  x : **Int** = y,
  y : **Int**

The scope of the value definition of x is equal to the the scope of the whole value declaration.
□

Visibility rules

The visibility rules are:

1. A definition is not visible outside its scope.
2. A definition is potentially visible throughout its scope. However, there may be places in the scope, where the definition is *hidden*, i.e. not visible. For instance, if the identifier or operator introduced by a definition is also introduced by another definition in an inner scope then the outer definition is hidden throughout the scope of the inner definition. For values the latter is only the case if the maximal types of the two values are undistinguishable. Other cases in which definitions are hidden are stated in the property sections.

Example 2.5

class
  variable
    v : **Bool** := **true**
  axiom local
        variable
          v : **Int** := 3
      in
        v = 7
      end
end

The scope of the variable definition ”v : **Bool** := **true**” is the whole class expression, while the scope of the local variable definition ”v : **Int** := 3” is the expression ”v = 7”. Therefore, according to visibility rule number 2, in the expression ”v = 7” only the local variable definition

is visible.
□

Example 2.6

```
class
  value
    v : Bool = true
  axiom
    local
      value
        v : Int = 3
    in
      v
    end
end
```

In the local expression the local value definition does not hide the outer value definition as the maximal types of the two value definitions are distinguishable. Therefore, both value definitions are visible in the local expression.
□

3 Overloading

An identifier or operator is said to be *overloaded* at a certain point if there are several definitions of that identifier or operator which are visible at that point.

Only value identifiers and operators are allowed to be overloaded.

Note that all operators have one or more predefined meanings which have the whole specification as scope. This implies that if the user defines an operator to have a maximal type distinguishable from the maximal types of the predefined meanings of the operator then in the scope of the user definition the operator is overloaded, cf. the visibility rules. If the user defines an operator to have a maximal type undistinguishable from one of the maximal types of the predefined meanings of the operator then this predefined meaning is hidden in the scope of the user-defined, cf. the visibility rules.

Overload resolution

For a specification to be useful there must be a unique legal interpretation of each identifier and operator, where we by an *interpretation* mean a corresponding definition. Now, an occurrence of an overloaded identifier or operator has several possible interpretations (namely one for each visible definition of it) and therefore the problem is to find its legal corresponding definition (if it has any).

Considering the context of the identifier or operator, some of the possible interpretations may be illegal according to the context conditions. In general the more context one considers the more information (context conditions) exists to identify illegal interpretations. But if the context considered is an expression which has the same maximal type for several different possible interpretations of the constituent overloaded identifiers and operators then further context will never make it possible to choose one of these interpretations over the other ones. Therefore all such interpretations are illegal.

In general the *legal* interpretations of the constituent identifiers and operators in a given context (a construct) are those

1. which satisfy the context conditions given by the construct, and
2. for which the construct has distinguishable maximal types if the construct is an expression (belongs to the syntactic category `expr`).

The overloading is said to be *resolvable* if there is exactly one legal interpretation of each identifier and operator in its innermost enclosing so-called "complete context".

A *complete context* is one of the following:

- The expr in a list_limitation.
- The expr in an explicit_let.
- The expr in a case_expr.
- A defined_item which is just an id_or_op.
- A specification.

Example 3.1

class
  value
    v : **Int**,
    v : **Bool**
  axiom
    v
end

The occurrence of v in the axiom is overloaded – it has two possible interpretations: either it is an integer or it is a boolean. However, only the latter interpretation satisfies the context condition that an axiom must have the maximal type **Bool**, and hence only this interpretation is legal.
□

Example 3.2

class
  value
    + : **Bool** × **Bool** → **Bool**,
    v : **Real**
  axiom
    true + **false** ≡ **true**
  axiom
    v ≡ 1.7 + 2.2
end

The two occurrences of the operator, +, in the axioms are overloaded – each of the occurrences has three possible interpretations: either it is the predefined integer addition (having the maximal type **Int** × **Int** $\stackrel{\sim}{\to}$ **Int**) or it is the predefined real addition (having the maximal type **Real** × **Real** $\stackrel{\sim}{\to}$ **Real**) or it is the user-defined boolean addition (having the maximal type **Bool** × **Bool** $\stackrel{\sim}{\to}$ **Bool**). Only the user-defined one satisfies the context conditions for the first occurrence, while only the predefined real addition satisfies the context conditions for the second occurrence.
□

Example 3.3

class
  value
    $+ : \textbf{Real} \times \textbf{Real} \to \textbf{Real}$
    $v : \textbf{Real}$
  axiom
    $v \equiv 1.7 + 2.2$
end

The occurrence of the operator, +, in the axiom has two possible interpretations: either it is the predefined integer addition (having the maximal type $\textbf{Int} \times \textbf{Int} \xrightarrow{\sim} \textbf{Int}$) or it is the user-defined real addition (having the maximal type $\textbf{Real} \times \textbf{Real} \xrightarrow{\sim} \textbf{Real}$). The predefined real addition is hidden since its maximal type is undistinguishable from the maximal type of the user-defined one. Only the user-defined real addition satisfies the context conditions, and hence only this interpretation is legal.

□

Example 3.4

value
  $v : \textbf{Int}$,
  $v : \textbf{Bool}$,
  $f : \textbf{Int} \to \textbf{Int}$,
  $f : \textbf{Bool} \to \textbf{Nat}$
axiom
  $f(v) \equiv 7$

There are two combinations of interpretations for f and v satisfying the context conditions. These are:

1. $f : \textbf{Int} \to \textbf{Int}$, $v : \textbf{Int}$
2. $f : \textbf{Bool} \to \textbf{Nat}$, $v : \textbf{Bool}$

However, for both combinations the maximal type of f(v) is the same, namely **Int**, and hence the expression f(v) has no legal interpretations.
□

Example 3.5

```
type
   B,
   C,
   A = B | C
value
   b : B,
   v : B,
   v : C,
   f : A → Bool,
   /* illegal */ a : A = v
axiom
   /* legal */ f(b),
   /* illegal */ f(v)
```

In the first axiom the identifier b has exactly one legal interpretation. Hence, the overloading is resolvable.

In the second axiom the identifier v has two possible interpretations: v : B and v : C. Both of these satisfy the context conditions, but for both the maximal type of the expression f(v) has the same maximal type. Therefore there are no legal interpretations of v in the expression f(v). Hence, the overloading is not resolvable.

In the definition of a the identifier v has two possible interpretations: v : B and v : C. Both of these satisfies the context conditions and are legal. Hence, the overloading is not resolvable.
□

4 Specifications

Syntax

specification ::=
  module_decl-string

module_decl ::=
  object_decl |
  scheme_decl

Terminology

A *module* is either an object or a scheme.

Meaning

A specification defines one or more modules.

Properties

In a specification the scope of the constituent module_decl-string is the module_decl-string itself. Note, that this means that the order of definitions is indifferent - an object or a scheme may be used before it is defined.

Context conditions

The constituent module_declss must be compatible, i.e. introduce distinct object and scheme identifiers.

4.1 Object declarations

Syntax

object_decl ::=
  object object_def-list

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

formal_array_parameter ::=
  [typing-list]

Terminology

An *object* is either a model or an array of models.

An *array of models* – also termed an *array* – is a mapping from values to models: each value is mapped to a single model.

The *index type* of an array is the type of values, all of which are mapped to a model by the array. An *index value* is a value within the index type.

An array maps any two distinct index values into two models that do not have variables or channels in common.

Meaning

An object declaration defines one or more objects.

- A model is defined by a definition of the form

  id : class_expr

  By this definition the identifier is bound to a model. The model is an arbitrary one belonging to the class represented by the class expression.

- An array of models is defined by a definition of the form

  id[typing_list] : class_expr

  By this definition the identifier is bound to an array of models. The index type of the array is the type represented by the typing list. Each index value belonging to the index type is mapped to a model. The model is an arbitrary one belonging to the class represented by the class expression – evaluated in the environment obtained by matching the index value against the decomposer also represented by the typing list.

  An array may be applied to an index value in an element object expression as described in section 4.4.2.

Any two defined objects do not have variables or channels in common.

Properties

In an object_def the scope of the opt-formal_array_parameter is the class_expr.

An object_def introduces the constituent id for an object. The object is an array if a formal_array_parameter is present else it is a model. If it is an array the maximal parameter type is the maximal type of the formal_array_parameter. The body is the constituent class_expr.

The maximal type of a formal_array_parameter is the maximal type of the single_typing the typing_list is a shorthand for.

Context conditions

In an object_decl the constituent object_defs must be compatible, i.e. introduce distinct identifiers.

4.2 Scheme declarations

Syntax

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

Terminology

A *scheme* is either a class or a parameterised class.

A *parameterised class* is a mapping from lists of objects to classes: each object list is mapped to a class.

Meaning

A scheme declaration defines one or more schemes.

- A class is defined by a definition of the form

  id = class_expr

  By this definition the identifier is bound to a class. The class is the one represented by the class expression.

- A parameterised class is defined by a definition of the form

  id(formal_scheme_argument_list) = class_expr

  By this definition the identifier is bound to a parameterised class.

  A parameterised class may be applied to a list of objects in a scheme instantiation as described in section 4.3.6. Under that section it is described which actual parameters are allowed and what the class resulting from the instantiation is.

Properties

In a scheme_def the scope of the opt-formal_scheme_parameter is the opt-formal_scheme_parameter itself and the class_expr.

A scheme_def introduces the constituent id for a scheme.

Context conditions

In a scheme_decl the constituent scheme_defs must be compatible, i.e. introduce distinct identifiers.

In a formal_scheme_parameter the constituent formal_scheme_arguments must be compatible, i.e. introduce distinct identifiers.

4.3 Class expressions

Syntax

class_expr ::=

basic_class_expr |
importing_class_expr |
extending_class_expr |
hiding_class_expr |
renaming_class_expr |
scheme_instantiation

Terminology

A *model* is an association of names with entities: each name is associated with a single entity. A model *provides* a name if it associates that name with an entity.

A model *satisfies* a definition if it provides the name introduced by that definition and if the entity associated with the name has the defined kind and if the properties stated in the definition *hold* in the model.

A *class* is a collection of models.

A name is *under-specified* if there exists at least two models in the class in which the name is associated with different entities. This corresponds to the case where the properties stated about the name are not *complete*.

Meaning

A class expression stands for a collection of definitions and represents the class consisting of all models that satisfy each of the definitions. Each model associates the identifiers and operators defined in the class expression with particular entities. For each alternative it is stated which definitions the class expression stands for.

4.3.1 Basic class expressions

Syntax

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

Meaning

A basic class expression stands for the definitions appearing in the declarations.

Properties

The immediate scope of the opt-decl-string is the opt-decl-string itself. Note, that this means that the order of definitions is indifferent.

Context conditions

The constituent decls must be compatible.

4.3.2 Importing class expression

Syntax

importing_class_expr ::=
  import object_expr-list **in** class_expr

Meaning

An importing class expression has the same meaning as the constituent class expression.

4.3.3 Extending class expressions

Syntax

extending_class_expr ::=
  extend class_expr-list **with** opt-decl-string **end**

Meaning

An extending class expression stands for the definitions which the class expressions stand for and the definitions which appear in the declarations.

Properties

The immediate scope of the opt-decl-string is the opt-decl-string itself. (Note, that this means that the order of definitions is indifferent.) The scopes of the class_exprs extend to the opt-decl-string.

Context conditions

The constituent class_exprs and decls must be compatible.

4.3.4 Hiding class expressions

Syntax

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

Meaning

A hiding class expression stands roughly speaking for the definitions that the constituent class expression stands for. The names that are mentioned in the defined item list can, however, not be referred to outside the class expression.

Properties

The scope of the class_expr extends to the id_or_ops in the defined_item-list, while all other definitions (than those of the class_expr) are hidden there. (From this and the visibility rules it follows that the defined_items must be defined in the class_expr.) The scope of the the definitions of the defined_items in the class_expr cannot be extended beyond the hiding_class_expr.

Context conditions

The constituent defined_items must be distinct.

A sort must not be hidden if it is used by a non-hidden entity.

4.3.5 Renaming class expression

Syntax

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

Meaning

A renaming class expression stands for the definitions that the constituent class expression stands for, but renamed according to the rename pairs in the renaming pair list.

Properties

The scope of the class_expr extends to the id_or_ops in the defined_items in the rename_pair-list, while all other definitions (than those of the class_expr) are hidden there. (From this and the visibility rules it follows that the defined (or old) items of the rename_pair-list must be defined in the class_expr.)

Context conditions

All new names must be distinct except if they are new names for values of distinguishable maximal types.

All old items of the rename_pair-list must be distinct. (In other words: there must not be more than one new name for each old item).

The new names must be different from the names of those old items (of the class_expr) which are not renamed, except for values, where a new name may be equal to the name of an old item, if the maximal type of the old item is distinguishable from the maximal type of the new.

4.3.6 Scheme instantiations

Syntax

scheme_instantiation ::=
  scheme-name opt-actual_scheme_parameter

actual_scheme_parameter ::=
  (object_expr-list)

Meaning

An instantiation is either an instantiation of a named class or of a named parameterised class.

- An instantiation of a named class has the form:

  name

  The *name* has the form *opt_qualification id* and the parameterised class must have been defined by a scheme definition (section 4.2) as follows:

  scheme
    id = body_class_expr

  The instantiation stands for the definitions that the body class expression stands for.

- An instantiation of a named parameterised class has the form:

  name(object_expr$_1$, ... ,object_expr$_n$)

  The *name* has the form *opt_qualification id* and the parameterised class must have been defined by a scheme definition (section 4.2) as follows:

  scheme
    id(
      id$_1$ opt_formal_array_parameter$_1$: class_expr$_1$, ... ,
      id$_n$ opt_formal_array_parameter$_n$: class_expr$_n$) =
        body_class_expr

  The instantiation stands for the definitions that the body class expression stands for – evaluated in an environment where each id_i has been bound to the object obtained by evaluating $object\_expr_i$.

Terminology

An object_expr-list is a *static implementation* of a formal_scheme_argument-list, if and only if:

- The number of the object_exprs is equal to the number of formal_scheme_arguments.

- Each of the object_exprs is a static implementation of the corresponding formal_scheme_argument.

An object_expr is a *static implementation* of a formal_scheme_argument if and only if:

- The object represented by the object_expr and the object defined by the formal_scheme_argument are either both arrays or both models.
- If they are both arrays then the maximal parameter types are the same.
- The body (a class_expr) of the object_expr is a static implementation of the body (a class_expr) of the formal_scheme_argument (which is an object_def.)

A class_expr is a *static implementation* of another (old) class_expr if and only if:

- For each non-axiom definition in the old class_expr there is a definition in the new class_expr of the same kind implementating it. (Here value definitions that are formed by multiple typings and product bindings, union definitions, short record definitions, variant definitions, multiple variable definitions and multiple channel definitions should be expanded to the collection of definitions they are shorthands for).

A type definition is a *static implementation* of another (old) type definition, if and only if:

- They introduce the same identifier.
- If the old type definition is an abbreviation definition then the new one is also an abbreviation definition and the maximal type of the new is equal to the maximal type of the old with all old sorts replaced by their corresponding new types.

A (single) value definition is a *static implementation* of another (old) (single) value definition, if and only if:

- They introduce the same name.
- The maximal type of the new is equal to the maximal type of the old with all old sorts replaced by their corresponding new types.

A (single) variable definition / (single) channel definition is a *static implementation* of another (old) (single) variable definition / (single) channel definition, if and only if:

- They introduce the same name.

- The maximal type of the new is equal to the maximal type of the old with all old sorts replaced by their corresponding new types.

An object definition is a *static implementation* of another (old) object definition, if and only if:

- They introduce the same name.
- They define either both an array or both a model.
- If they define arrays then they have the same maximal parameter type.
- The class expression of the new object definition is a static implementation of the class expression of the old object definition.

A scheme definition is a *static implementation* of another (old) scheme definition, if and only if:

- They introduce the same name.
- They have the same number of formal scheme arguments.
- The maximal parameter types of the formal scheme arguments of the old scheme definition are equal of the corresponding maximal parameter types of the new scheme definition.
- The class expressions of the formal scheme arguments of the old scheme definition are static implementations of the corresponding class expressions of the new scheme definition.
- The class expression of the new scheme definition is a static implementation of the class expression of the old scheme definition.

Context conditions

In a scheme_instantiation the name must represent a scheme. There must be an actual_scheme_parameter present if and only if the scheme is parameterised, (i.e. a formal_scheme_parameter is present in the corresponding scheme_def). If an actual_scheme_parameter is present then the constituent object_expr-list must be a static implementation of the formal_scheme_argument-list (of the formal_scheme_parameter) of the corresponding scheme definition. The definitions introduced by the scheme_instantiation must not contain type cycles and must be compatible.

In an actual_scheme_parameter any two object_exprs must not provide the same variable or channel.

4.4 Object expressions

Syntax

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

Meaning

An object expression represents an object.

Properties

An object_expr has an associated class expression, called the *body*. If it represents an array, then it also has an associated *maximal parameter type*.

4.4.1 Names

Properties

The body of a name is the class_expr of the corresponding object definition. The maximal parameter type of a name representing an array is the maximal type of the formal_array_parameter of the corresponding object definition.

Context conditions

For an object_expr being a name, this name must represent an object.

4.4.2 Element object expressions

Syntax

element_object_expr ::=

array-object_expr actual_array_parameter

actual_array_parameter ::=
[*pure*-expr-list]

Meaning

An element object expession represents a model obtained as follows. The object expression represents an array and the actual array parameter represents the value obtained by evaluating the expression list as a product. This value should be an index value of the array. The model is obtained by applying the array to the value (i.e. the model is that model which the value is mapped to by the array).

Properties

The body is the body of the constituent object_expr. Note, that in the context conditions, two applications of the same array to distinct actual array parameters are considered as defining the same entities.

The maximal type of an actual_array_parameter
of the form [e_1] is t_1
and of the form [e_1, ..., e_n] is $\mathrm{t}_1 \times ... \times \mathrm{t}_n$,
where t_1, ..., t_n are the maximal types of the constituent exprs e_1, ..., e_n.

Context conditions

In an element_object_expr the constituent object_expr must represent an array.

In an element_object_expr the maximal type of the actual_array_parameter must be less than or equal to the maximal parameter type of the object_expr.

The exprs in the actual_array_parameter must be pure.

4.4.3 Array object expressions

Syntax

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

Meaning

An array object expression represents an array. The index type of the array is the type represented by the typing list. Each index value belonging to the index type is mapped to a model. This model is the model obtained by evaluating the element object expression in the environment obtained by matching the index value against the decomposer also represented by the typing list.

Properties

The scope of the constituent typings is the object_expr.

The maximal parameter type is the maximal type of the single_typing the constituent typing-list is a shorthand for. The body is the body of the constituent object_expr.

Context conditions

The object_exp must represent a model.

4.4.4 Fitting object expressions

Syntax

fitting_object_expr ::=
  object_expr renaming

Meaning

A fitting object expression represents the object represented by the object expression, but with provided names renamed according to the renaming.

In case the object represented by the object expression is an array, the names in each of the models which the index values are mapped to are renamed.

Properties

In a fitting_object_expr the scope of the body of the constituent object_expr extends to the id_or_ops in the defined_items in the renaming, while all other definitions (than those of the body of the constituent object_expr) are hidden there. (From this and the visibility rules it follows that the defined (or old) items of the renaming must be defined in the body of the object_expr.)

If a fitting_object_expr represents an array the maximal parameter type is the maximal parameter type of constituent object_expr. The body is the body of the constituent object_expr renamed according to the constituent renaming.

4.5 Renamings

Syntax

renaming ::=
  { rename_pair-list }

rename_pair ::=
  defined_item **for** defined_item

defined_item ::=
  id_or_op |
  disambiguated_item

disambiguated_item ::=
  id_or_op : type_expr

Terminology

If a rename_pair occurs in the renaming of a fitting_object_expr then the name on the right-hand side of **for** is called a *new* name and the name on the left-hand side of **for** is called an *old* name. If it occurs in a rename_class_expr then the name on the left-hand side of **for** is called a *new* name and the name on the right-hand side of **for** is called an *old* name.

To *rename* something according to a rename_pair means to replace all occurrences of the old name with the new name.

Meaning

A renaming represents the combination of the renamings represented by each rename pair in the rename pair list.

The type expression within a disambiguated item is useful when the name due to overloading represents several values with different types. The type expression then identifies precisely one of the values.

Context conditions

In a rename_pair there must not be a type_expr in that defined_item which contains a new name.

In a renaming all new names must be distinct except if they are new names for values of distinguishable maximal types.

In a renaming all old items must be distinct. (In other words: there must not be more than one new name for each old item).

In a disambiguated_item the id_or_op must represent a value and its maximal type and the maximal type of type_expr must be the same.

5 Declarations

Syntax

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

Terminology

A declaration is a list of definitions all of the same *kind* – scheme, object, type, value, variable, channel or axiom. Each definition normally introduces a name for an *entity* of that kind, and one or more *properties*.

Object and scheme declarations are described in sections 4.1 and 4.2.

5.1 Type declarations

Syntax

type_decl ::=
  type commented_type_def-list

commented_type_def ::=
  opt-comment-string type_def

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

Meaning

A type declaration defines one or more types and zero or more values.

Context conditions

The type names introduced in the constituent type_defs must be distinct from each other and the introduced value names.

The value names introduced in the constituent type_defs must be distinct unless their maximal types are distinguishable.

5.1.1 Sort definitions

Syntax

sort_def ::=
  id

Terminology

A *sort* – or synonymously *abstract type* – is a type with no predefined operations for generating and manipulating its values.

Meaning

A sort definition defines a sort by just giving its name.

Since a sort is not born with predefined operations for generating and manipulating its values, the writer of a specification must define these as values him- or herself. Their definition indirectly states properties about the sort. If for example two values of the same sort are defined and they are required to be different, then indirectly the sort is required to contain at least two values.

Properties

The maximal type of a sort is the sort itself.

5.1.2 Variant definitions

Syntax

variant_def ::=
  id == variant-choice

variant ::=
  constant_variant |
  record_variant

constant_variant ::=
  constructor opt-subtype_naming

record_variant ::=
  constructor component_kinds opt-subtype_naming

constructor ::=
  id_or_op |
  _

component_kinds ::=
  (component_kind-list)

component_kind ::=
  opt-destructor type_expr opt-reconstructor

destructor ::=
  id_or_op :

reconstructor ::=
  ↔ id_or_op

subtype_naming ::=
  @ id

Meaning

A variant definition is a shorthand for writing a name for an abstract type, names for its constructors and names for destructors and reconstructors. Additionally it generally provides an implicit induction axiom. We shall in the following describe how a variant definition of the form

type

id == variant$_1$| ... |variant$_n$

represents a sort definition, some subtype definitions, some value definitions and some axioms. We will deal in turn with constructors, subtypes, destructors, reconstructors and induction axioms. We will also use the following example throughout.

```
type
  Tree ==
    empty @ Empty_Tree |
    node(
      left : Tree,
      val : Elem ↔ repl_value,
      right : Tree) @ Non_Empty_Tree
```

- *Constructors*

  Tree has two variants, one of which is constant and one a record variant. It has two subtype namings.

  The subtype naming is dealt with below, and so we will ignore it here. We also deal with destructors and reconstructors below, and so ignore them here.

  We can now construct a series of declarations that are equivalent to the original. Firstly, we have an abstract type declaration for the variant type being defined:

  ```
  type
    id
  ```

  For our example this declaration would be

  ```
  type
    Tree
  ```

  Secondly, for each variant, indexed i say, which is not a wildcard, we obtain a value declaration.

  If the variant is a constant variant, say con_i, we obtain the value declaration

  value
  con$_i$: id

  which simply says that con_i is a (constant) value of type *id*. For our example we would have the single value declaration

  ```
  value
    empty : Tree
  ```

If the variant is a record variant having n_i components, say,

$\mathrm{con}_i(\mathrm{T}_{i,1}, ..., \mathrm{T}_{i,n_i})$

we obtain the value declaration

value
$\mathrm{con}_i : \mathrm{T}_{i,1} \times ... \times \mathrm{T}_{i,n_i} \to \mathrm{id}$

which says that con_i is a total function from the product of its component types to the type *id*. con_i constructs values of type *id* from values of its component types, (as indicated by the name, "constructor", of its syntactic category).

For our example we would have the single value declaration

value
node : Tree × Elem × Tree → Tree

Constant and record variants which have wildcards for their constructors do not generate any value declarations (except for any destructors or reconstructors attached to the components of a record variant).

- *Subtypes*

  Any variant can have a subtype name (an identifier, $type\_id_i$) associated with it. This will generate an additional type declaration as follows

  - for a non-wildcard constant variant

    type
    $\mathrm{type\_id}_i = \{|\ \mathrm{x} : \mathrm{id} \bullet \mathrm{x} = \mathrm{id}_i\ |\}$

  - for a non-wildcard record variant

    type
    $\mathrm{type\_id}_i =$
    $\{|\ \mathrm{x} : \mathrm{id} \bullet$
    $\exists\ \mathrm{x}_1 : \mathrm{T}_{i,1}, ..., \mathrm{x}_{n_i} : \mathrm{T}_{i,n_i} \bullet$
    $\mathrm{x} = \mathrm{con}_i(\mathrm{x}_1, ..., \mathrm{x}_{n_i})\ |\}$

  - for a constant or record wildcard variant

    type
    $\mathrm{type\_id}_i = \{|\ \mathrm{x} : \mathrm{id} \bullet \mathrm{p(x)}\ |\}$
    value
    p : id → **Bool**

    The identifier *p* is under-specified. It must not be already in scope, and should be hidden at the level of the smallest enclosing class expression.

    Our example introduces two subtype names which generates the subtype declarations

type
  Empty_tree = {| x : Tree • x = empty |},
  Non_empty_tree =
    {| x : Tree •
      $\exists$ x_1 : Tree, x_2 : Elem, x_3 : Tree •
        x = node(x_1,x_2,x_3) |}

- *Destructors*

  Each destructor $dest_{i,j}$ introduced in a record variant

  $\mathrm{con}_i(..., \mathrm{dest}_{i,j} : \mathrm{T}_{i,j}, ...)$

  generates firstly a value declaration

  value
    $\mathrm{dest}_{i,j} : \mathrm{id} \xrightarrow{\sim} \mathrm{T}_{i,j}$

  For our example we would have the following declarations

  value
    left : Tree $\xrightarrow{\sim}$ Tree,
    val : Tree $\xrightarrow{\sim}$ Elem,
    right : Tree $\xrightarrow{\sim}$ Tree

  If a constructor is present, it also generates an axiom of the following form

  axiom
    $\forall\ \mathrm{x}_1 : \mathrm{T}_{i,1}, ..., \mathrm{x}_{n_i} : \mathrm{T}_{i,n_i}$ •
      $\mathrm{dest}_{i,j}(\mathrm{con}_i(\mathrm{x}_1, ..., \mathrm{x}_{n_i})) \equiv \mathrm{x}_j$

  For our example we would have the axioms

  axiom
    $\forall$ x_1 : Tree, x_2 : Elem, x_3 : Tree •
      left(node(x_1, x_2, x_3)) = x_1,
    $\forall$ x_1 : Tree, x_2 : Elem, x_3 : Tree •
      val(node(x_1, x_2, x_3)) = x_2,
    $\forall$ x_1 : Tree, x_2 : Elem, x_3 : Tree •
      right(node(x_1, x_2, x_3)) = x_3,

- *Reconstructors*

  Each reconstructor $recon_{i,j}$ introduced in a record variant

  $\mathrm{con}_i(..., ...\ \mathrm{T}_{i,j} \leftrightarrow \mathrm{recon}_{i,j}, ...)$

generates firstly a value declaration

value
  $\text{recon}_{i,j} : \text{T}_{i,j} \times \text{id} \xrightarrow{\sim} \text{id}$

For our example we would have, for the one reconstructor *repl_val*

value
  $\text{repl\_val} : \text{Elem} \times \text{Tree} \xrightarrow{\sim} \text{Tree}$

If there are destructors associated with the variant then for each destructor $dest_{i,k}$ there is an axiom relating it to the reconstructor $recon_{i,j}$. For the case when j and k are equal we obtain the axiom

$$\forall\ \text{x}_j : \text{T}_{i,j},\ \text{x} : \text{id} \bullet$$
$$\text{dest}_{i,j}(\text{recon}_{i,j}(\text{x}_j,\text{x})) \equiv \text{x}_j$$

which expresses the fact that a destructor recovers the component value changed by a corresponding reconstructor.

When j and k are different we obtain the axiom

$$\forall\ \text{x}_j : \text{T}_{i,j},\ \text{x} : \text{id} \bullet$$
$$\text{dest}_{i,k}(\text{recon}_{i,j}(\text{x}_j,\text{x})) \equiv \text{dest}_{i,k}(\text{x})$$

which expresses the fact that changing a component value by a reconstructor does not affect other components.

In our example, we obtain the following three axioms

axiom
  $\forall\ \text{x}_2 : \text{Elem},\ \text{x} : \text{Tree} \bullet$
    $\text{left}(\text{repl\_val}(\text{x}_2,\text{x})) \equiv \text{left}(\text{x})$,
  $\forall\ \text{x}_2 : \text{Elem},\ \text{x} : \text{Tree} \bullet$
    $\text{val}(\text{repl\_val}(\text{x}_2,\text{x})) \equiv \text{x}_2$,
  $\forall\ \text{x}_2 : \text{Elem},\ \text{x} : \text{Tree} \bullet$
    $\text{right}(\text{repl\_val}(\text{x}_2,\text{x})) \equiv \text{right}(\text{x})$

- *Induction axioms*

  Provided there are no wildcard variants in our type definition we will also obtain an induction axiom. (The removal of the induction axiom is the main reason for using wildcard variants — they allow us to later add further variants, or components of variants, and obtain implementation. If there were an induction axiom, making such additions would negate it and so could not give implementation.)

  If there is no recursion in the type, i.e. none of the component types in any variants involve *id*, then the induction axiom is simple:

$$
\begin{array}{l}
\forall\ f : id \rightarrow \mathbf{Bool} \bullet \\
\quad (\\
\qquad (\forall\ x_1 : T_{1,1},\ ...,\ x_{n_1} : T_{1,n_1} \bullet \\
\qquad\quad f(con_1(x_1,...,x_{n_1}))) \\
\qquad \wedge\ ...\ \wedge \\
\qquad (\forall\ x_1 : T_{n,1},\ ...,\ x_{n_n} : T_{n,n_n} \bullet \\
\qquad\quad f(con_n(x_1,...,x_{n_n}))) \\
\quad) \Rightarrow \\
\qquad \forall\ x : id \bullet f(x)
\end{array}
$$

(In this definition, for any variant, index i, say, that is constant we take n_i to be zero so that the quantification in the conjunct disappears and we obtain a conjunct $f(con_i)$.)

Suppose now that the type is recursive, and that the j'th component in the i'th variant is *id*. Then in the above definition the i'th conjunct becomes

$$
\begin{array}{l}
(\forall\ x_1 : T_{i,1},\ ...,\ x_j : id,\ ...,\ x_{n_i} : T_{i,n_i} \bullet \\
\quad f(x_j) \Rightarrow f(con_i(x_1,...,x_j,...,x_{n_i})))
\end{array}
$$

There are obvious extensions to this when there are two or more component types in a variant equal to *id*. For two such we would get a conjunct of the form

$$
\begin{array}{l}
(\forall\ x_1 : T_{i,1},\ ...,\ x_j : id,\ ...,\ x_k : id,\ ...,\ x_{n_i} : T_{i,n_i} \bullet \\
\quad (f(x_j) \wedge f(x_k)) \Rightarrow \\
\qquad f(con_i(x_1,...,x_j,...,x_k,...,x_{n_i})))
\end{array}
$$

This is the case in our example, which has the induction axiom

axiom

$$
\begin{array}{l}
\forall\ f : Tree \rightarrow \mathbf{Bool} \bullet \\
\quad (\\
\qquad f(empty) \\
\qquad\quad \wedge \\
\qquad (\forall\ x_1 : Tree,\ x_2 : Elem,\ x_3 : Tree \bullet \\
\qquad\quad (f(x_1) \wedge f(x_3)) \Rightarrow f(node(x_1,x_2,x_3))) \\
\quad) \Rightarrow \\
\qquad \forall\ x : Tree \bullet f(x)
\end{array}
$$

So to prove some property of the type *Tree* we prove it for *empty* and we prove it for a constructed node assuming it is true for the left and right subtrees.

Another extension is when *id* is a component of $T_{i,j}$ instead of equal to it. For instance, suppose $T_{i,j}$ is $U \times id$. Then the conjunct would be

$$
\begin{array}{l}
(\forall\ x_1 : T_{i,1},\ ...,\ (y,z) : (U \times id),\ ...,\ x_{n_i} : T_{i,n_i} \bullet \\
\quad f(z) \Rightarrow f(con_i(x_1,...,(y,z),...,x_{n_i})))
\end{array}
$$

This leads us to the possibility that *id* is a component of a variant type $T_{i,j}$ and hence to the problem of mutually recursive variant types. The general rule here is fairly complicated, and the reader is referred to the proof rules ([2]) for its formulation. We will present an example. Suppose we generalise trees to have lists of subnodes, and define lists by variants. Then we might have the definitions

```
type
  Tree ==
    empty_tree |
    node(
      val : Elem,
      sub : List),
  List ==
    empty_list |
    list(
      head : Tree,
      tail : List)
```

The induction axiom for these is a joint one, formulated as follows:

```
axiom
  ∀ tf : Tree → Bool, lf : List → Bool •
    (
      tf(empty_tree)
        ∧
      (∀ x1 : Elem, x2 : List •
        lf(x2) ⇒ tf(node(x1,x2)))
        ∧
      lf(empty_list)
        ∧
      (∀ x1 : Tree, x2 : List •
        (tf(x1) ∧ lf(x2)) ⇒ lf(list(x1,x2)))
    ) ⇒
      ∀ x1 : Tree, x2 : List • (tf(x1) ∧ lf(x2))
```

So to prove a pair of properties of *Tree* and *List* we prove the appropriate properties for the constants *empty_tree* and *empty_list*, and also prove them for the constructed values assuming the appropriate properties of components.

Properties

The maximal type of the type being defined is the type itself.

The constituent constructors, destructors and reconstructors introduce names of values. The maximal types of these are the maximal types of the types given in the meaning section.

The constituent subtype_namings introduces names for types. The maximal type of these is the variant type.

Context conditions

The name of the variant type being defined and all names of the subtypes introduced in the constituent subtype_namings must be distinct from each other and from the names in the constituent constructors, destructors and reconstructors.

The constructors, deconstructors and reconstructors must only have the same name if their maximal types are distinguishable.

5.1.3 Union definitions

Syntax

union_def ::=
  id = *type*-name-choice2

Meaning

A union definition is a shorthand for writing a variant definition including constructors and destructors. A union definition of the form

type
  id = opt_qualification$_1$ id$_1$| ... | opt_qualification$_n$ id$_n$

is a shorthand for

type
  id ==
    id$_1$_to_id(id$_1$_from_id : opt_qualification$_1$ id$_1$)| ... |
    id$_n$_to_id(id$_n$_from_id : opt_qualification$_n$ id$_n$)

In addition, in contexts where an expression is required to have a type which is "less than or equal to" *id*, any expression, say *expr*, having one of the types id_i $(1 \leq i \leq n)$ is allowed - it is not necessary to apply the corresponding constructor, $id_i\_to\_id$, in order to get an expression of type *id*. Loosely, *expr* is a shorthand for writing $id_i\_to\_id(expr)$. This may be generalized. See section 6, where the ordering "less than" is defined.

Properties

The maximal type of the type being defined is the type itself.

The implicit constructors and destructors introduce names of values. The maximal types of these are given by the variant_def the union_def is a shorthand for.

Context conditions

The constituent names must represent types and the last ids in the names must be distinct.

5.1.4 Short record definitions

Syntax

short_record_def ::=
  id :: component_kind-string

Meaning

A short record definition is a shorthand for a variant definition with a single variant including a constructor. A short record definition of the form

type
  id :: component_kind_1 ... component_kind_n

is a shorthand for

type
  id == mk_id(component_kind_1, ... ,component_kind_n)

Properties

The maximal type of the type being defined is the type itself.

The implicit constructor, the constituent destructors and the reconstructors introduce names of values. The maximal type of these are given by the variant_def the short_record_def is a shorthand for.

Context conditions

The name of the short record type being defined must be distinct from any name in the constituent destructors and reconstructors.

The implicit constructor, the deconstructors and the reconstructors must only have the same name if their maximal types are distinguishable.

5.1.5 Abbreviation definitions

Syntax

abbreviation_def ::=
  id = type_expr

Meaning

An abbreviation definition introduces a name for the type represented by the type expression.

Properties

The maximal type of the type being defined is the maximal type of the constituent type_expr.

Context conditions

The type being defined must not be among the types that the right-hand side of the abbreviation_def depends on. In this case the definition would be illegally cyclic.

5.2 Value declarations

Syntax

value_decl ::=
  value commented_value_def-list

commented_value_def ::=

opt-comment-string value_def

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

Meaning

A value declaration defines one or more values.

A typing introduces one or more identifiers and/or operators for values of particular types.

Context conditions

The value names introduced in the constituent value_defs must be distinct unless their maximal types are distinguishable.

5.2.1 Explicit value definitions

Syntax

explicit_value_def ::=
single_typing = *pure*-expr

Meaning

An explicit value definition of the form

value
single_typing = expr

is a shorthand for

value

  single_typing
axiom
  E(single_typing) = expr

The meta function

  E : single_typing → expr

turns the binding part of a single typing into an expression simply by copying identifiers, commas and parentheses and by bracketting operators. The type part of the single typing is ignored by the function. For example,

  E((x,+) : T) = (x,(+))

Context conditions

The maximal type of the expr must be less than or equal to the maximal type of the single_typing.

The constituent expr must be pure.

5.2.2 Implicit value definitions

Syntax

implicit_value_def ::=
  single_typing *pure*-restriction

Meaning

An implicit value definition of the form

value
  single_typing • expr

is a shorthand for

value
single_typing
axiom
expr

Context conditions

The restriction must be pure.

5.2.3 Explicit function definitions

Syntax

explicit_function_def ::=
single_typing formal_function_application ≡ 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

pre_condition ::=
pre *readonly_logical*-expr

Meaning

An explicit function definition of the form

value

single_typing
formal_function_application ≡ expr opt_pre_condition

is a shorthand for

value
single_typing
axiom
Q(formal_function_application)(D)
E(formal_function_application) ≡ expr opt_pre_condition

Two meta functions E and Q have been applied here. The function

Q : formal_function_application → type_expr → 'an optional quantification'

extracts a parameter quantification from a formal function application. The second argument to the function is a type expression representing the domain type of the function being defined (how to obtain D should be obvious in the individual case). The function works as follows

Q(id())(D) =
'no quantification'
Q(id(binding))(D) =
∀ binding : D •
Q(id(binding_1,...,binding_n))(D) =
∀ (binding_1,...,binding_n) : D •
Q(op id) =
∀ id : D •
Q(id_1 op id_2) =
∀ (id_1,id_2) : D •

The function

E : formal_function_application → expr

turns a formal function application into an expression simply by copying identifiers, operators, commas and parentheses – bracketting operators in case of an id_application. For example,

E(f(x,+)) = f(x,(+))
E(**card** s) = **card** s
E(x + y) = x + y

Properties

In an explicit_function_def the scope of the formal_function_application is expr and opt-pre_condition.

The context of a formal_function_application determines a maximal context type for the formal_function_application.

In an explicit_function_def the maximal context type of the formal_function_application is the maximal type of the single_typing.

In a formal_function_parameter of the form ($b_1, \ldots, b_n$) the maximal context types of the constituent bindings $b_1, \ldots, b_n$ are $t_1, \ldots, t_n$, respectively, where $t_1 \times \ldots \times t_n$ is the parameter part of the maximal context type (a function type).

The maximal type of the id introduced in a prefix_application is the parameter type part, t_1, of the maximal context type, $t_1 \stackrel{\sim}{\rightarrow}$ acc t_2.

The maximal types of the first and second id in an infix_application are t_1 and t_2, respectively, where the maximal context type is $t_1 \times t_2 \stackrel{\sim}{\rightarrow}$ acc t_3.

Context conditions

The binding in the single_typing must be an id_or_op (i.e. not a product_binding).

If the id_or_op in the single_typing is an identifier then the formal_function_application must be an id_application of this identifier.

If the id_or_op in the single_typing is a prefix operator then the formal_function_application must be a prefix_application of this prefix operator.

If the id_or_op in the single_typing is an infix operator then the formal_function_application must be an infix_application of this infix operator.

The maximal type of the single_typing must be a function type. If the formal_function_application is an infix_application then the parameter part of the function type must be a product type of length 2. If the formal_function_application is an id_application then the function type must be curried at least as many times as there are formal_function_parameters.

I.e. there are the three following forms, where some of the arrows may be substituted with $\stackrel{\sim}{\rightarrow}$:

$$\begin{array}{l} \mathrm{id} : t_1 \rightarrow \mathrm{acc}_1\ t_2\ \ldots \rightarrow \mathrm{acc}_n\ t_{n+1} \\ \mathrm{id}(\mathrm{id}_1)(\mathrm{id}_2)\ldots(\mathrm{id}_n) \equiv \mathrm{expr\ opt{-}pre\_condition} \end{array}$$

$$\mathrm{prefix\_op} : t_1 \rightarrow \mathrm{acc}\ t_2$$

prefix_op id ≡ expr opt−pre_condition

infix_op : $t_1 \times t_2 \rightarrow$ acc t_3
id infix_op id′ ≡ expr opt−pre_condition

The maximal type of the expr must be less than or equal to the type one obtains from the maximal type of the single_typing, when one strips as many parameter types as there are formal_function_parameters in the formal_function_application.

The expr and the opt-pre_condition must only access those variables and channels that are accessible according to the access descriptors acc_1 - acc_n in the first form example above and the access descriptor acc in the two last form examples.

In an id_application, the identifiers introduced in the constituent formal_function_parameters must be distinct unless they have distinguishable maximal types. In an infix_application the two identifiers must be distinct unless they have distinguishable maximal types.

In a pre_condition the expr must be readonly and must have the maximal type **Bool**.

5.2.4 Implicit function definitions

Syntax

implicit_function_def ::=
  single_typing formal_function_application post_condition opt-pre_condition

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

result_naming ::=
  as binding

Meaning

An implicit function definition of the form

value
  single_typing
    formal_function_application post_condition opt_pre_condition

is a shorthand for

value
  single_typing
axiom
  Q(formal_function_application)(D)
    E(formal_function_application) post_condition opt_pre_condition

The two functions E and Q and the type expression D have been defined above.

Properties

In an implicit_function_def the scope of the formal_function_application is the post_condition and opt-pre_condition.

The context of a post_condition determines a maximal context type for the post_condition.

In an implicit_function_def the maximal context type of the formal_function_application is the maximal type of the single_typing and the maximal context type of the post_condition is the result part of the type one obtains from the maximal type of the single_typing, when one strips as many parameter types as there are formal_function_parameters in the formal_function_application.

In a post_condition the scope of the opt-result_naming is the expr.

In a result_naming the maximal context type of the binding is the maximal context type of the innermost enclosing post_condition.

Context conditions

The binding in the single_typing must be an id_or_op (i.e. not a product_binding).

If the id_or_op in the single_typing is an identifier then the formal_function_application must be an id_application of this identifier.

If the id_or_op in the single_typing is a prefix operator then the formal_function_application must be a prefix_application of this prefix operator.

If the id_or_op in the single_typing is an infix operator then the formal_function_application must be an infix_application of this infix operator.

The maximal type of the single_typing must be a function type. If the formal_function_application is an infix_application then the parameter part of the function type must be a product type of length 2. If the formal_function_application is an id_application then the function type must be curried at least as many times as there are formal_function_parameters.

I.e. there are the three following forms, where some of the arrows may be substituted with $\stackrel{\sim}{\rightarrow}$:

$\text{id} : \text{t}_1 \rightarrow \text{acc}_1\ \text{t}_2\ ...\ \rightarrow \text{acc}_n\ \text{t}_{n+1}$
$\text{id}(\text{id}_1)(\text{id}_2)...(\text{id}_n)$ post_condition opt−pre_condition

prefix_op : $\text{t}_1 \rightarrow \text{acc}\ \text{t}_2$
prefix_op id post_condition opt−pre_condition

infix_op : $\text{t}_1 \times \text{t}_2 \rightarrow \text{acc}\ \text{t}_3$
id infix_op id′ post_condition opt−pre_condition

The post_condition and the opt-pre_condition must only read those variables and channels that are accessible according to the access descriptors acc_1 - acc_n in the first form example above and the access descriptor acc in the two last form examples.

In a post_condition the expr must be readonly and must have the maximal type **Bool**.

5.3 Variable declarations

Syntax

variable_decl ::=
  variable commented_variable_def-list

commented_variable_def ::=
  opt-comment-string variable_def

variable_def ::=
  single_variable_def |
  multiple_variable_def

single_variable_def ::=
  id : type_expr opt-initialisation

initialisation ::=
  := *pure*-expr

multiple_variable_def ::=
  id-list2 : type_expr

Terminology

A *variable* is a container capable of holding values of a particular type. All variables are kept in the so-called *state*.

The contents of a variable can be changed explicitly by an assignment expression. It can thus change contents within its lifetime. Variables are in particular used to define functions with side-effects on some global state.

Meaning

A single variable definition defines a variable and its associated type. In addition, an initialisation expression can be given, the value of which is the initial value of the variable. The initial value is the value kept in the variable 'when its surrounding class expression is instantiated' and restored to it by an initialise expression (section 7.19).

If no initialisation is given, the initial value of the variable is just some arbitrarily chosen value within its type.

A multiple variable definition is just a shorthand for two or more single variable definitions. A multiple variable definition of the form

variable
  id_1, ... ,id_n : type_expr

is a shorthand for

variable
  id_1 : type_expr,
  ⋮
  id_n : type_expr

Properties

In a variable_def the maximal types of the constituent ids is the maximal type of the type_expr.

Context conditions

The names introduced in the constituent variable_defs must be distinct.

The expr in an initialisation must be pure.

5.4 Channel declarations

Syntax

channel_decl ::=
  channel commented_channel_def-list

commented_channel_def ::=
  opt-comment-string channel_def

channel_def ::=
  single_channel_def |
  multiple_channel_def

single_channel_def ::=
  id : type_expr

multiple_channel_def ::=
  id-list2 : type_expr

Terminology

A *channel* is a medium that concurrently executing expressions can communicate through.

In order for two expressions to communicate through a channel, one expression must offer an output communication to the channel whilst the other expression must offer an input communication from the channel.

Communication is *synchronized*: the outputting expression only outputs to the channel if the inputting expression simultaniously inputs from the channel.

Meaning

A single channel definition defines a channel and its associated type. All values communicated over the channel must have this type.

A multiple channel definition is just a shorthand for two or more single channel definitions. A multiple channel definition of the form

channel
  id_1, ... ,id_n : type_expr

is a shorthand for

channel
  id_1 : type_expr,
  ⋮
  id_n : type_expr

Properties

In a channel_def the maximal types of the constituent ids is the maximal type of the type_expr.

Context conditions

The names introduced in the constituent channel_defs must be distinct.

5.5 Axiom declarations

Syntax

axiom_decl ::=
  axiom opt-axiom_quantification axiom_def-list

axiom_quantification ::=
  forall typing-list •

axiom_def ::=
  opt-comment-string opt-axiom_naming *pure_logical*-expr

axiom_naming ::=
  [id]

Meaning

An axiom_decl defines one or more axioms, each being a boolean expr defining additional properties of names introduced by other definitions.

An axiom can be given a name (axiom_naming), but such a naming does not add to the properties.

Each axiom is implicitly prefixed with '□'. That is, an axiom_decl of the form

axiom
  opt_axiom_naming$_1$ expr$_1$,
  ⋮
  opt_axiom_naming$_n$ expr$_n$

is a shorthand for

axiom
  opt_axiom_naming$_1$ □ expr$_1$,
  ⋮
  opt_axiom_naming$_n$ □ expr$_n$

An axiom_quantification is a shorthand for a distributed quantification. That is, an axiom_decl of the form

axiom forall typing_list •
  opt_axiom_naming$_1$ expr$_1$,
  ⋮
  opt_axiom_naming$_n$ expr$_n$

is a shorthand for

axiom
  opt_axiom_naming$_1$ □ ∀ typing_list • expr$_1$,
  ⋮
  opt_axiom_naming$_n$ □ typing_list • expr$_n$

Properties

In an axiom_decl the scope of the opt-axiom_quantification is the axiom_def-list.

Context conditions

The names introduced in the constituent axiom_defs must be distinct.

The expr in an axiom_def must be readonly and have the maximal type **Bool**.

6 Type expressions

Syntax

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

Terminology

A *type* is a set of values. We distinguish between three kinds of types:

- *predefined types* are represented by literals build into the language. These types include for example the integers and the booleans.
- *abstract types* are represented by sort names defined in sort definitions (section 5.1.1).
- *compound types* are build from other types by application of a *type operator* to one or more types.

A type T_1 is a *subtype* of a type T_2 if T_1 is a subset of T_2.

A type is said to be *maximal* if (it can be representated by a type expression and) it is not a subtype of any other type (which can be represented by a type expression).

The *maximal type* of a type is the maximal type of which it is a subtype.

Two types are *undistinguishable* if their maximal types only differ wrt. constituent access descriptors in constituent function types.

Two types are *distinguishable* if they are not undistinguisable.

The union definitions in a specification give rise to an ordering on types, called *less than.* This ordering is defined inductively by the following rules:

1. Each type, t, represented on the right-hand side of a union definition is less than the type, t', represented on the left-hand side.
2. If t is less than t' and t' is less than t" then t is less than t". (The ordering is transitive.)
3. If t is less than t' and acc' contains all the accesses of acc then
   (a) $\ldots \times$ t $\times \ldots$ is less than $\ldots \times$ t' $\times \ldots$
   (b) t-infset is less than t'-infset
   (c) t^ω is less than t'^ω
   (d) t $\underset{m}{\rightarrow}$ s is less than t' $\underset{m}{\rightarrow}$ s, where s is some type
   (e) s $\underset{m}{\rightarrow}$ t is less than s $\underset{m}{\rightarrow}$ t', where s is some type
   (f) s $\overset{\sim}{\rightarrow}$ acc t is less than s $\overset{\sim}{\rightarrow}$ acc' t', where s is some type

A type, t, is said to be an *upper bound* of a collection of types if all types in the collection is less than or equal to t.

A type is said to be a *least upper bound* of a collection of types if it is an upper bound of this collection and it is less than all other upper bounds.

Meaning

A type expression represents a type.

Properties

A type_expr has an associated maximal type, which is equal to the maximal type of the type it represents.

6.1 Names

Context conditions

For a type_expr being a name, the name must represent a type.

6.2 Type literals

Syntax

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

Meaning

A type literal represents a predefined type. There are the following literals:

  Unit =
    { () }
  Bool =
    { **true**,**false** }
  Int =
    { ...,-2,-1,0,1,2,... }
  Nat =
    {| i : **Int** • i ≥ 0 |}
  Real =
    { ...,-4.3,...,12.23,... }
  Text =
    Char*
  Char =
    { ′a′,′b′,... }

The unit value '()' represents the single value in type **Unit**. The natural numbers **Nat** is a subtype of the integers **Int**. Characters **Char** are the ASCII characters. Texts **Text** are just character lists. A Text literal in double quotes (″...″) is a shorthand for a list of characters. That is, a text of the form

  ″c_1 ... c_n″

is a shorthand for

  ⟨′c_1′,...,′c_n′⟩

Properties

The maximal type of **Unit** is **Unit**.
The maximal type of **Bool** is **Bool**.
The maximal type of **Int** is **Int**.
The maximal type of **Nat** is **Int**.
The maximal type of **Real** is **Real**.
The maximal type of **Text** is $\mathbf{Char}^{w}$.
The maximal type of **Char** is **Char**.

6.3 Product type expressions

Syntax

product_type_expr ::=
  type_expr-product2

Terminology

A *product* is a value of the form

$(v_1,...,v_n)$

The *length* of a product_type_expr is the number of constituent type_exprs.

Meaning

A product type expression of the form

$\text{type\_expr}_1 \times ... \times \text{type\_expr}_n$

represents the type of all products of the form

$(v_1,...,v_n)$

where each v_i has the type represented by $\textit{type\_expr}_i$.

Properties

The maximal type of a product_type_expr of the form $t_1 \times ... \times t_n$ is $t'_1 \times ... \times t'_n$, where $t'_1, \ldots, t'_n$ are the maximal types of the constituent type_exprs $t_1, \ldots, t_n$.

6.4 Set type expressions

Syntax

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**

Terminology

A *set* is an unordered collection of distinct values of the same type.

Meaning

A set type expression represents a type of subsets of the type represented by the constituent type expression. If the type operator is **-set**, the type will contain all finite subsets. If the type operator is **-infset**, the type will contain all (infinite as well as finite) subsets.

Properties

The maximal type of a set_type_expr of the form t-**set** or t-**infset** is t'-**infset**,
where t' is the maximal type of the constituent type_expr t.

6.5 List type expressions

Syntax

list_type_expr ::=
  finite_list_type_expr |
  infinite_list_type_expr

finite_list_type_expr ::=
  type_expr*

infinite_list_type_expr ::=
  type_expr$^\omega$

Terminology

A *list* is an ordered sequence of values of the same type, possibly including duplicates.

Meaning

A list type expression represents a type of lists of elements of the type represented by the constituent type expression. If the type operator is *, the type will contain all finite lists. If the type operator is $^\omega$, the type will contain all (infinite as well as finite) lists.

Properties

The maximal type of a list_type_expr of the form t^* or t^ω is t'^ω,
where t' is the maximal type of the constituent type_expr t.

6.6 Map type expressions

Syntax

map_type_expr ::=
  type_expr $\underset{m}{\rightarrow}$ type_expr

Terminology

A map can be conceived as a (possibly infinite) collection of pairs (v_1, v_2) where v_1 is a domain value, v_2 is a range value and v_1 is related to v_2. The *domain* of a map is the set of values, v_1, for which there exists a value, v_2, such that (v_1, v_2) is in the map. The *range* of a map is the set of values, v_2, for which there exists a value, v_1, such that (v_1, v_2) is in the map.

A map is said to be *non-deterministic* if it relates a domain value with more than one range value.

A map can be applied to a value in its domain to find a corresponding value in the range – arbitrarily chosen among the possible in the non-deterministic case.

Meaning

A map type expression represents the type of all maps, each of which has a subset of the first type as domain and a subset of the second type as range.

Properties

The maximal type of a map_type_expr of the form $\mathrm{t}_1 \underset{m}{\overrightarrow{}} \mathrm{t}_2$ is $\mathrm{t'}_1 \underset{m}{\overrightarrow{}} \mathrm{t'}_2$,
where $\mathrm{t'}_1$ and $\mathrm{t'}_2$ are the maximal types of the constituent type_exprs t_1 and t_2.

6.7 Function type expressions

Syntax

function_type_expr ::=
  type_expr function_arrow result_desc

function_arrow ::=
  $\stackrel{\sim}{\rightarrow}$ |
  $\rightarrow$

result_desc ::=
  opt-access_desc-string type_expr

Terminology

A *function* maps the values of one type – the *parameter type* – into values of another type – the *result type*. In addition a function can, when applied:

- Access variables by reading from them or writing to them.
- Access channels by inputting from them or outputting to them.

An *applicative function* is a function that does not access variables or channels. An *imperative function* is a function that accesses variables or channels.

An *operation* is an imperative function that does not access channels, only variables. A *process* is an imperative function that accesses channels.

A function's *extended parameter type* can be defined as:

- its parameter type,
- the variables it can read from together with their types,
- the channels it can input from together with their types.

A function's *extended result type* can be defined as:

- its result type,
- the variables it can write to together with their types,
- the channels it can output to together with their types.

In general, if a function is applied to a value, evaluated in a state and inputs values from channels, then if all of these satisfy the subtype constraints of the extended parameter type, then the subtype constraints of the extended result type will be satisfied. Otherwise, one can only be guaranteed that the maximal type constraints of the extended result type are satisfied.

Meaning

A function type expression represents a type of functions from the parameter type represented by the type expression to the result type represented by the type expression in the result description. The access description string specifies which accesses to variables and channels the functions are allowed when applied. Depending on the function arrow the functions are either partial or total as described below.

- *Partial functions*

  A partial function type expression of the form

  type_expr$_1$ $\stackrel{\sim}{\rightarrow}$ opt_access_desc_string type_expr$_2$

  defines a set of functions, f, such that for any x belonging to the maximal type of *type_expr*$_1$, the application

  f(x)

  is an expression that might deadlock, default or diverge at any stage in its execution.

  If

  - x : *type_expr*$_1$,
  - and the current state satisfies the subtype constraints on the variables,
  - and any inputs satisfy the subtype constraints on the channels,

  then any outputs will satisfy the subtype constraints on the channels.

  If additionally the application expression terminates – possibly after some sequence of communication, then

  - the resulting value will be within *type_expr*$_2$,
  - and the resulting state will satisfy the subtype constraints on the variables.

  Alternatively, if

  - *not*(x : *type_expr*$_1$) but x belonging to the maximal type of *type_expr*$_1$,
  - or the current state does not satisfy the subtype constraints on the variables,
  - or some inputs do not satisfy the subtype constraints on the channels,

  then the subtype constraints on outputs, resulting value and resulting state cannot be guaranteed. The maximal type constraints will, however, still be guaranteed.

- *Total functions*

  A total function type expression of the form

  type_expr$_1$ $\rightarrow$ opt_access_desc_string type_expr$_2$

  defines the subset of the partial functions f:

  type_expr$_1$ $\stackrel{\sim}{\rightarrow}$ opt_access_desc_string type_expr$_2$

  such that if

  - x : *type_expr*$_1$,

– and the current state satisfies the subtype constraints on the variables,
– and any inputs satisfy the subtype constraints on the channels,

then the application

f(x)

is a total expression. This can also be expressed by defining total functions as a subtype of partial functions. A type expression of the form

type_expr → result_desc

is a shorthand for

{| f : type_expr $\stackrel{\sim}{\rightarrow}$ result_desc •
∀ x : type_expr • f(x) **post true** |}

Properties

The maximal type of a function_type_expr of the form $t_1 \stackrel{\sim}{\rightarrow}$ acc t_2 or $t_1 \rightarrow$ acc t_2 is $t'_1 \stackrel{\sim}{\rightarrow}$ acc t'_2, where t'_1 and t'_2 are the maximal types of the constituent type_exprs t_1 and t_2.

6.7.1 Access descriptions

Syntax

access_desc ::=
access_mode access-list

access_mode ::=
read |
write |
in |
out

access ::=
variable_or_channel-name |
completed_access |
comprehended_access

completed_access ::=
opt-qualification **any**

comprehended_access ::=
{ access-list | *pure*-set_limitation }

Meaning

A function type expression describes what global variables and channels can be accessed from the functions it represents. For each variable and channel it is additionally described in what way it can be accessed.

An access description describes the variables/channels having a particular access-mode. A variable can be given access-mode:

- **read** if it may only be read from.
- **write** if it may be written to, that is: changed by an assignment; variables with this access-mode have automatically **read** access-mode.

A channel can be given access-mode:

- **in** if it may be input from.
- **out** if it may be output to.

An access describes a logically related set of variables/channels having the access-mode. An access list gives access to the union of the individual accesses. An access can besides a name be a:

- **completed access**

  A completed access gives access to all variables/channels introduced by a particular model.

  If the qualification is absent, all variables/channels in the innermost enclosing model are given access to, except that completed accesses within schemes refer to the model at the point of instantiation, not that at the point of definition.

  If the qualification is present, it represents a model. All the variables/channels in that model are then given access to.

- **comprehended access**

  A comprehended access gives access to the set of variables/channels obtained as follows: for each environment in the set of environments represented by the set limitation, the union of the acesses in the access list is obtained. The result is the union of all these unions.

Context conditions

For an access being a name, the name must represent

- a variable if it occurs in the access-list of an access_descr having **read** or **write** as access_mode
- a channel if it occurs in the access-list of an access_descr having **in** or **out** as access_mode

In a comprehended_access the set_limitation must be pure.

6.8 Subtype expressions

Syntax

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

Meaning

A subtype expression represents a subtype of the type represented by the single typing. The subtype contains any value that makes the restriction hold – evaluated in the environment obtained by matching the value against the decomposer also represented by the single typing.

Properties

The maximal type of a subtype_expr is the maximal type of the constituent single_typing.

Context conditions

The restriction must be pure.

6.9 Bracketted type expressions

Syntax

bracketted_type_expr ::=
  (type_expr)

Meaning

A bracketted type expression represents the same type as represented by the type expression.

Properties

The maximal type of a bracketted_type_expr is the maximal type of the constituent type_expr.

7 Expressions

Syntax

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 |
  bracketted_expr |
  infix_expr |
  prefix_expr |
  comprehended_expr |
  initialise_expr |
  assignment_expr |
  input_expr |
  output_expr |
  structured_expr

Terminology

An expression is evaluated – or synonymously 'executed' – in a state to yield a value. In addition, the expression may

- read the value of variables,
- write to variables by assignments,
- input from channels,
- output to channels.

More formally, the *effect* of an expression – evaluated in a state – is to offer zero or more communications on channels and then to do one of the following

- *terminate* succesfully thereby returning a value and a possibly changed state. B
- *deadlock* when it has as its only possible effect to communicate with the surroundings through a channel that is concealed from (not visible in) the surroundings. Such a situation can for example arise when the expression locally declares a channel and then tries to communicate with the surroundings through it.
- *default* when its effect can never be selected in an internal choice.
- *diverge* when it has as a possible effect to continue executing without terminating, without attempting to input from or output to channels, and without deadlocking or defaulting.

An expression is *non-deterministic* if it may choose internally between a set of possible effects. Two evaluations of the expression in the same state might thus give two different effects. As a consequence, the eventually returned value can be one of a set of possible values. Likewise the returned state can be one of a set of possible states.

An expression is *converging* if it has as the only possible behaviour to terminate succesfully.

An expression is *total* if

- there is no possibility of deadlock, default or divergence at any stage in the execution of the expression (even after some sequence of communications),
- if at any stage in the execution of the expression it comes to an end then the result returned is deterministic,

An expression may be build from sub-expressions. Unless otherwise stated the following holds:

1. Any sub-expression may be non-deterministic and this non-determinism propagates such that the whole expression gets non-deterministic.
2. Any sub-expression may be diverging, deadlocking or defaulting. The diverging, deadlocking or defaulting of the sub-expression propagates such that the whole expression becomes diverging, deadlocking or defaulting.

An expr is said to *access* a variable if it reads (from) or writes to it.

An expr is said to *access* a channel if it inputs from or outputs to it.

An expr is said to be *pure* if it does not access any variable or channel.

An expr is said to be *readonly* if it does not write to any variable and it does not input from or output to any channel.

Two exprs are said to access the state *independently* if each of the exprs does not read any of the variables which are written in by the other expr.

Two exprs are said to be *parallelizable* if they access the state independently.

An expr is said to *offer a communication*, if it inputs from or outputs to a channel.

Properties

An expr has an associated maximal type (such that if the expr terminates successfully then its value belongs to its maximal type).

An expr also has an associated description of which variables it reads, which variables it writes to, which channels it inputs from and which channels it outputs to. In this description the following conventions has been used: If an expr writes to a variable, then it also reads that variable.

7.1 Value literals

Syntax

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

unit_literal ::=
  ( )

bool_literal ::=
  true |
  false

Meaning

The effect of a value literal is to return the value represented by the literal.

Properties

The maximal type of a unit_literal is **Unit**.
The maximal type of a bool_literal is **Bool**.
The maximal type of a int_literal is **Int**.
The maximal type of a real_literal is **Real**.
The maximal type of a text_literal is $\mathbf{Char}^{\omega}$.
The maximal type of a char_literal is **Char**.

A value_literal does not access any variables or channels.

7.2 Names

Properties

A name representing a value does not access any variables or channels. A name representing a variable reads that variable.

Context conditions

For an expr being a name, the name must represent a value or a variable.

7.3 Pre names

Syntax

pre_name ::=
  variable-name `

Meaning

A pre_name occurs within a post_condition, see section 7.13. The effect of a pre_name is to return the contents in the pre-state of the variable represented by name.

Properties

The local variable definitions of the innermost enclosing post_condition are hidden in the name.

The maximal type of a pre_name is the maximal type of the constituent name.

A pre_name reads the variable it represents.

Context conditions

A pre_name must occur within a post_condition.

The name must represent a variable.

7.4 Basic expressions

Syntax

basic_expr ::=
  chaos |
  skip |
  stop |
  swap

Meaning

- The effect of **chaos** is to diverge.
- The effect of **skip** is to return the unit value of type **Unit**.
- The effect of **stop** is to deadlock.
- The effect of **swap** is to default.

Properties

The maximal type of a basic_expr is **Unit**.

A basic_expr does not access any variables or channels.

7.5 Product expressions

Syntax

product_expr ::=
  (expr-list2)

Meaning

The effect of a product expression is the effect of the expression list evaluated as a product.

Properties

The maximal type of a product_expr of the form (e_1, ...,e_n) is $t_1 \times ... \times t_n$,
where $t_1, \ldots, t_n$ are the maximal types of the constituent exprs $e_1, \ldots, e_n$.

A product_expr accesses any of the variables and channels which the constituent exprs access.

Context conditions

The constituent exprs must access the state independently and at most one of them may offer a communication.

7.6 Set expressions

Syntax

set_expr ::=
  ranged_set_expr |
  enumerated_set_expr |
  comprehended_set_expr

7.6.1 Ranged set expressions

Syntax

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

Meaning

The effect of a ranged set expression is to return a set of integers in a range delimited by a lower bound and an upper bound.

The first constituent expression is evaluated to return the lower bound i_1 and the second constituent expression is evaluated to return the upper bound i_2. The set contains all integers i such that $i_1 \leq i \leq i_2$.

Properties

The maximal type of a ranged_set_expr is **Int-infset**.

A ranged_set_expr accesses any of the variables and channels which the constituent exprs access.

Context conditions

The constituent exprs must be readonly and must have the maximal type **Int**.

7.6.2 Enumerated set expressions

Syntax

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

Meaning

The effect of an enumerated set expression is to return a set of explicitely specified values.

The set contains all elements in the list returned by the expression list – evaluated as a list.

Properties

The maximal type of an enumerated_set_expr having one or more constituent exprs is t-**infset**, where t is the least upper bound of the maximal types of the constituent exprs. The maximal type of an enumerated_set_expr having no constituent exprs is t-**infset**, where t is free to be any type.

An enumerated_set_expr accesses any of the variables and channels which the constituent exprs access.

Context conditions

The constituent exprs must be readonly.

The maximal types of the constituent exprs must have a least upper bound.

7.6.3 Comprehended set expressions

Syntax

comprehended_set_expr ::=
  { *readonly*-expr | set_limitation }

set_limitation ::=
  typing-list opt-restriction

restriction ::=
  • *readonly_logical*-expr

Meaning

The effect of a comprehended set expression is to return a set, the elements of which are obtained by evaluating the constituent expression in all those environments that satisfies a certain restriction.

For each environment in the set of environments represented by the set limitation (see below), the expression is evaluated. If the expression is convergent and deterministic, the returned value is included in the set. In the case of non-convergence or non-determinism, the particular evaluation does not contribute with a set member. A comprehended set expression is convergent and deterministic.

A set limitation represents a subset of the environments that the typing list represents: those that makes the restriction hold. An absent restriction is equivalent to the restriction • **true**.

We say that a restriction holds if it is convergent and deterministic and returns the value **true**.

Properties

In a comprehended_set_expr the scope of the set_limitation extends to the constituent expr.

In a set_limitation the immediate scope of the typings is the constituent restriction.

The maximal type of a comprehended_set_expr is t-**infset**, where t is the maximal type of the constituent expr.

A comprehended_set_expr reads any of the variables and channels which the constituent expr and set_limitation read.

A set_limitation reads any of the variables and channels which the constituent restriction reads.

A restriction reads any of the variables and channels which the constituent expr reads.

Context conditions

In a comprehended_set_expr the constituent expr must be readonly.

In a restriction the constituent expr must be readonly and must have the maximal type **Bool**.

7.7 List expressions

Syntax

list_expr ::=
  ranged_list_expr |
  enumerated_list_expr |
  comprehended_list_expr

7.7.1 Ranged list expressions

Syntax

ranged_list_expr ::=
  ⟨ *readonly_integer*-expr .. *readonly_integer*-expr ⟩

Meaning

The effect of a ranged list expression is to return a list of integers in a range delimited by a lower bound and an upper bound.

The two constituent expressions are evaluated to return the integers i_1 – the value of the first constituent expression – and i_2 – the value of the second constituent expression. The list contains all integers i such that $i_1 \leq i \leq i_2$, occurring in increasing order.

Properties

The maximal type of a ranged_list_expr is $\mathbf{Int}^{\omega}$.

A ranged_list_expr reads any of the variables and channels which the constituent exprs read.

Context conditions

The constituent exprs must be readonly and must have maximal type **Int**.

7.7.2 Enumerated list expressions

Syntax

enumerated_list_expr ::=
  ⟨ *readonly*-opt-expr-list ⟩

Meaning

The effect of an enumerated list expression is the effect of the expression list evaluated as a list.

Properties

The maximal type of an enumerated_list_expr having one or more constituent exprs is t^ω, where t is the least upper bound of the maximal types of the constituent exprs.

The maximal type of an enumerated_list_expr having no constituent exprs is t^ω, where t is free to be any type.

An enumerated_list_expr reads any of the variables and channels which the constituent exprs read.

Context conditions

The constituent exprs must be readonly.

The maximal types of the constituent exprs must have a least upper bound.

7.7.3 Comprehended list expressions

Syntax

comprehended_list_expr ::=
  ⟨ *readonly*-expr | list_limitation ⟩

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

Meaning

The effect of a comprehended list expression is to return a list generated on the basis of another list.

For each environment in the list of environments represented by the list limitation (see below) – processed from left to right – the constituent expression is evaluated. The returned value is included in the list at the corresponding position.

A list limitation evaluates to a list of environments as follows. The constituent expression returns a list. Each list element in the list – processed from left to right – is then matched against the binding to obtain an environment. In case this environment makes the restriction hold, the environment is included in the resulting environment list at the corresponding position.

An absent restriction is equivalent to the restriction • **true**.

Properties

In a comprehended_list_expr the scope of the list_limitation extends to the constituent expr.

In a list_limitation the immediate scope of the binding is the constituent restriction.

The maximal type of a comprehended_list_expr is t^ω, where t is the maximal type of the constituent expr.

In a list_limitation the maximal context type of the constituent binding is t, where t^ω is the maximal type of the constituent expr.

A comprehended_list_expr reads any of the variables and channels which the constituent expr and list_limitation read.

A list_limitation reads any of the variables and channels which the constituent expr and restriction read.

Context conditions

In a comprehended_list_expr the constituent expr must be readonly.

In a list_limitation the constituent expr must be readonly and must have a maximal type which is a list type.

7.8 Map expressions

Syntax

map_expr ::=
  enumerated_map_expr |
  comprehended_map_expr

7.8.1 Enumerated map expression

Syntax

enumerated_map_expr ::=
  [opt-expr_pair-list]

expr_pair ::=
  readonly-expr ↦ *readonly*-expr

Meaning

The effect of an enumerated map expression is to return a map of explicitly specified pairs.

Each expression pair in the expression pair list represents a pair of values. The map contains all the pairs represented by such expression pairs. In case the expression pair list is empty the map is empty.

Properties

The maximal type of an enumerated_map_expr having one or more constituent expr_pairs is $t_1 \xrightarrow[m]{} t_2$, where t_1 is the least upper bound of the domain types and t_2 is the least upper bound of the range types of the constituent expr_pairs.

The maximal type of an enumerated_map_expr having no constituent exprs is $t_1 \xrightarrow[m]{} t_2$, where t_1 and t_2 are free to be any types.

The maximal domain type and the maximal range type of an expr_pair are the maximal types of the first and the second constituent expr, respectively.

An enumerated_map_expr reads any of the variables and channels which the constituent expr_pairs read.

An expr_pair reads any of the variables and channels which the constituent exprs read.

Context conditions

In an enumerated_map_expr the maximal domain types of the the constituent expr_pairs must have a least upper bound and the maximal range types of the the constituent expr_pairs must have a least upper bound.

In an expr_pair the exprs must be readonly.

7.8.2 Comprehended map expressions

Syntax

comprehended_map_expr ::=
  [expr_pair | set_limitation]

Meaning

The effect of a comprehended map expression is to return a map the pairs of which are obtained by evaluating the expression pair in all those environments that satisfies a certain restriction.

For each environment in the set of environments represented by the set limitation, the expression pair is evaluated. If the expression pair is convergent and deterministic, the resulting value pair is included in the map. In the case of non-convergence or non-determinism, the particular evaluation does not contribute with a pair. A comprehended map expression is convergent and deterministic.

Properties

In a comprehended_map_expr the scope of the set_limitation extends to the constituent expr_pair.

The maximal type of a comprehended_map_expr is $t_1 \underset{m}{\rightarrow} t_2$, where t_1 is the maximal domain type and t_2 is the maximal range type of the constituent expr_pair.

A comprehended_map_expr reads any of the variables and channels which the constituent expr_pair and set_limitation read.

7.9 Function expressions

Syntax

function_expr ::=
  λ lambda_parameter • expr

lambda_parameter ::=
  lambda_typing |

single_typing

lambda_typing ::=
(opt-typing-list)

Meaning

The effect of a function expression is to return a function. The lambda parameter represents:

- a type – the parameter type of the function,
- a decomposer – against which an actual parameter is matched to yield a parameter environment mapping formal parameter identifiers and operators to actual parameter values.

In the case of a single typing the type and decomposer are those represented by the single typing. In the case of a lambda typing, the type and decomposer are those represented by the optional typing list.

When the function is applied to an actual parameter p within the parameter type, p is matched against the decomposer to yield an environment in which the body expression is evaluated.

Properties

In a function_expr the scope of the lambda_parameter is expr.

The maximal type of a function_expr is $\mathrm{t}_1 \stackrel{\sim}{\rightarrow} \mathrm{acc}\ \mathrm{t}_2$, where t_1 is the maximal type of the lambda_parameter, t_2 is the maximal type of the expr and acc is a description of which variables and channels the expr accesses.

The maximal type of an lambda_parameter is given for each of its alternatives.

The maximal type of a lambda_typing is **Unit** if there is no typing-list present else it is the maximal type of the typing the typing-list is a shorthand for.

A function_expr does not access any variables or channels.

7.10 Application expressions

Syntax

application_expr ::=

list_or_map_or_function-expr actual_function_parameter-string

actual_function_parameter ::=
(opt-expr-list)

Meaning

The effect of an application expression is obtained by applying a function, a map or a list to an actual parameter.

An application of the form where there is only one actual function parameter:

expr actual_function_parameter

represents the application of a function, map or list to an actual parameter. The expression and actual function parameter are evaluated to return respectively an applicable value and an actual parameter. The actual parameter is the value returned by the optional expression list evaluated as a product.

In the case the applicable value is a:

- function – this is just applied to the actual parameter.
- map – the returned value is non-deterministically chosen between those mapped to by the actual parameter in the map. If the actual parameter maps to no values in the map , the application defaults.
- list – the actual parameter must be a positive number between one and the length of the list. In that case, the list element at that position becomes the value returned. In case the actual parameter does not indicate a position in the list, the application diverges.

An application expression of the form

expr actual_function_parameter$_1$ ... actual_function_parameter$_n$

is equivalent to

(...(expr actual_function_parameter$_1$)...) actual_function_parameter$_n$

Properties

The properties of an application_expr are given for the case, where there is only one actual_function_parameter. The properties for the case where there are more than one actual_function_parameter is given by the properties of its equivalent application_expr (see above).

The maximal type of an application_expr is determined by the maximal type of the constituent expr. If this is:

- a function type, $t_1 \stackrel{\sim}{\rightarrow} acc\ t_2$, then it is the result type, t_2,
- list type, t^ω, then it is the element type, t,
- a map type, $t_1 \underset{m}{\rightarrow} t_2$, then it is the range type t_2.

An application_expr accesses any of the variables and channels which the constituent expr and actual_function_parameter access and if the expr has a maximal type which is a function type, $t_1 \stackrel{\sim}{\rightarrow} acc\ t_2$, then also any of the variables and channels which the function body accesses as described in acc.

Context conditions

The context conditions of an application_expr are given for the case, where there is only one actual_function_parameter. The context conditions for the case where there are more than one actual_function_parameter is given by the context conditions of its equivalent application_expr (see above).

In an application_expr the maximal type of the expr must be a function type, a list type or a map type. Furthermore, if the maximal type of the expr is:

- a function type, $t_1 \stackrel{\sim}{\rightarrow} acc\ t_2$, then the maximal type of the actual_function_parameter must be less than or equal to the type t_1
- a list type, t^ω, then the maximal type of the actual_function_parameter must be equal to **Int**
- a map type, $t_1 \underset{m}{\rightarrow} t_2$, then the maximal type of the actual_function_parameter must be less than or equal to the type t_1

In an application_expr the following constructs must access the state independently and at most one of them may offer a communication: the expr and the actual_function_parameter and if the expr has a maximal type which is a function type, $t_1 \stackrel{\sim}{\rightarrow} acc\ t_2$, then also the function body (the access of which is described in acc).

In an actual_function_parameter the constituent exprs must access the state independently and at most one of them may offer a communication.

7.11 Quantified expressions

Syntax

quantified_expr ::=
  quantifier typing-list restriction

quantifier ::=
  ∀ |
  ∃ |
  ∃!

Meaning

The effect of a quantified expression is to return a boolean value depending on the value returned by a predicate for each environment in a set of environments. The typing list represents a set of environments. In case the quantifier is:

∀ , the returned value is **true** iff. the restriction holds for all the environments.

∃ , the returned value is **true** iff. the restriction holds for at least one of the environments.

∃! , the returned value is **true** iff. the restriction holds for exactly one of the environments.

A quantified expression is convergent and deterministic.

Properties

In a quantified_expr the scope of the constituent typings is the restriction.

The maximal type of a quantified_expr is **Bool**.

A quantified_expr reads any of the variables and channels which the constituent restriction reads.

7.12 Equivalence expressions

Syntax

equivalence_expr ::=
  expr ≡ expr opt-pre_condition

Meaning

The effect of an equivalence_expr is to return a boolean value that depends on whether the two exprs yield the same effect, when each is evaluated in the current state. The returned value is **true** if and only if at least one of the following two conditions are satisfied

- The pre_condition, if present, does not hold.

  A pre_condition holds in a given state if the constituent expr is convergent and deterministic and returns the value **true**.

- The equivalence holds.

  The equivalence holds if and only if the first expr evaluated in the current state represents exactly the same effect as the second expr evaluated in the same state. That is, the two exprs must represent the same effect concerning non-determinism, state-modification, external communication and returned value, but also the same effect concerning divergence, deadlock and defaulting.

An equivalence_expr is convergent and deterministic.

Properties

The maximal type of an equivalence_expr is **Bool**.

An equivalence_expr reads any of the variables which the constituent exprs read.

Context conditions

The maximal types of the the constituent exprs must have a least upper bound.

7.13 Post expressions

Syntax

post_expr ::=
  expr post_condition opt-pre_condition

Meaning

The effect of a post_expr is to return a boolean value that depends on the effect of expr when evaluated in the current state, the pre-state. The effect of expr is only used to determine this boolean value and is ignored thereafter.

The value returned by the post_expr is **true** if and only if one or more of the following conditions are satisfied

- The pre_condition, if pressent, does not hold in the pre-state. A pre_condition holds in a given state if the constituent expr is convergent and deterministic and returns the value **true**.
- The subtype constraints on variables are not satisfied in the pre-state.
- Inputs performed within expr do not satisfy subtype constraints on channels.
- The post_condition holds.

The post_condition holds if and only if all of the following conditions are satisfied

- The expr is total.
- Any outputs performed within expr satisfy the subtype constraints on the channels.
- If expr terminates:
  - the returned state, the post-state, satisfies the subtype constraints on the variables,
  - the value returned – matched against the post_condition result_naming if present – and the post-state make the post-condition expr hold: it is convergent and deterministic and returns the value **true**.

    Within the post-condition expr, variables in the pre-state can be referred to by suffixing them with a hook (pre_name). Variables of the post-state are accessed through their normal (un-hooked) names.

A post_expr is convergent and deterministic.

Properties

The maximal type of a post_expr is **Bool**.

The maximal context type for the post_condition is the maximal type of the constituent expr.

A post_expr reads any of the variables or channels which the constituent expr reads.

7.14 Disambiguation expressions

Syntax

disambiguation_expr ::=
  expr : type_expr

Meaning

The effect of a disambiguation expression is the effect of the constituent expression. Due to overloading the constituent expression may represent many values with different types. The type expression identifies exactly one of these values.

Properties

The maximal type of a disambiguated_expr is the maximal type of the type_expr.

A disambiguated_expr accesses any of the variables and channels which the constituent expr accesses.

Context conditions

The maximal type of the expr must be less than or equal to the maximal type of the type_expr.

7.15 Bracketted expressions

Syntax

bracketted_expr ::=

(expr)

Meaning

The effect of a bracketted expression is the effect of the constituent expression.

7.16 Infix expressions

Syntax

infix_expr ::=
  stmt_infix_expr |
  axiom_infix_expr |
  value_infix_expr

7.16.1 Statement infix expressions

Syntax

stmt_infix_expr ::=
  expr infix_combinator expr

Meaning

See the definition of infix combinators.

Properties

For the infix_combinator being

$\Box$, $\sqcap$: the maximal type of the stmt_infix_expr is the least upper bound of the maximal types of the constituent exprs.

$\|$, $\#$: the maximal type of the stmt_infix_expr is **Unit**.

; : the maximal type of the stmt_infix_expr is the maximal type of the second constituent expr.

A statement_infix_expr accesses any of the variables and channels which the two constituent exprs access.

Context conditions

For the infix_combinator being

$\sqcap$, $\sqcap\!\!\!\!\mid$: the maximal types of the two exprs must have a least upper bound.

$\|$, $\nparallel$: the two exprs must have the maximal type **Unit** and must be parallelizable.

; : the first expr must have the maximal type **Unit**.

7.16.2 Axiom infix expressions

Syntax

axiom_infix_expr ::=
  logical-expr infix_connective *logical*-expr

Meaning

See the definition of infix connectives.

Properties

The maximal type of an axiom_infix_expr is **Bool**.

An axiom_infix_expr accesses any of the variables and channels which the two constituent exprs access.

Context conditions

The two exprs must have the maximal type **Bool**.

7.16.3 Value infix expressions

Syntax

value_infix_expr ::=
  expr infix_op expr

Meaning

The effect of a value infix expression is to return the value obtained by applying the infix operator to a pair of values. Each of the constituent expressions are evaluated to return the values v_1 – the value of the first constituent expression – and v_2 – the value of the second constituent expression. The infix operator is then applied to the pair (v_1, v_2).

Properties

The maximal type of a value_infix_expr is the result type part of the maximal type of the infix_op.

A value_infix_expr accesses any of the variables and channels which the two constituent exprs access and any of the variables and channels which the function body accesses as described in the access description part of the maximal type of the infix_op.

Context conditions

The type $\mathrm{t}_1 \times \mathrm{t}_2$, where t_1 and t_2 are the maximal types of the two exprs, must be less than or equal to the parameter type part of the maximal type of the infix_op. The two exprs and the function body (the access of which is described in the access description part of the maximal type of the infix_op) must access the state independently and at most one of them may offer a communication.

7.17 Prefix expressions

Syntax

prefix_expr ::=
  axiom_prefix_expr |
  value_prefix_expr

7.17.1 Axiom prefix expressions

Syntax

axiom_prefix_expr ::=
  prefix_connective *logical*-expr

Meaning

See the definition of prefix connectives.

Properties

The maximal type of an axiom_prefix_expr is **Bool**.

An axiom_prefix_expr does not access any variables or channels if the prefix_connective is □, else it accesses any of the variables and channels which the constituent expr accesses.

Context conditions

The expr must have the maximal type **Bool**.

If the constituent prefix_connective is □ then the constituent expr must be readonly.

7.17.2 Value prefix expressions

Syntax

value_prefix_expr ::=
  prefix_op expr

Meaning

The effect of a value prefix expression is to return the value obtained by applying the prefix operator to the value returned by the constituent expression.

Properties

The maximal type of a value_prefix_expr is the result type part of the maximal type of the prefix_op.

A value_prefix_expr accesses any of the variables and channels which the constituent expr accesses and any of the variables and channels which the function body accesses as described in the access description part of the maximal type of the prefix_op.

Context conditions

The maximal type of the expr must be less than or equal to the parameter part of the maximal type of the prefix_op.

The expr and the function body (the access of which is described in the access description part of the maximal type of the prefix_op) must access the state independently and at most one of them may offer a communication.

7.18 Comprehended expressions

Syntax

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

Meaning

The effect of a comprehended expression is obtained by applying a binary infix combinator to a set of expressions instead of to just two expressions. This has the straight-forward explanation since the infix combinators possible here are all commutative and associative:

$$\Box \quad \sqcap \quad \|$$

The set contains an expression for each environment in the set of environments represented by the set limitation. The expression is evaluated in that environment and in the current state.

In the case the set contains a single expression, the comprehended expression represents the same effect as the expression. In the case the set is empty we have:

$\|$ {} ≡ **stop**
$\sqcap$ {} ≡ **swap**
$\|$ {} ≡ **skip**

Properties

In a comprehended_expr the scope of the set_limitation extends to the expr.

The maximal type of a comprehended_expr is the maximal type of the expr.

A comprehended_expr accesses any of the variables and channels which the constituent expr and set_limitation access.

Context conditions

The infix_combinator must be associative and commutative, that is, it must be one of the following: $\|$, $\|$, $\sqcap$.

For the infix_combinator, $\|$, the expr must have the maximal type **Unit**.

7.19 Initialise expressions

Syntax

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

Meaning

The effect of an initialise expression is to re-assign to selected variables their initial values. The value returned by the initialise expression is the unit value. The initial value of a variable is given in connection with its definition.

In case the qualification is absent, all variables introduced by the innermost enclosing model are initialised, except that initialisations within schemes refer to the model at the point of instantiation, not that at the point of definition.

In the case the qualification is present, it represents a model. All the variables in that model are then initialised.

Properties

The maximal type of an initialise_expr is **Unit**.

An initialise_expr writes to all variables initialised by it.

7.20 Assignment expressions

Syntax

assignment_expr ::=
  variable-name := expr

Meaning

The effect of an assignment expression is to assign the value of the constituent expression to the variable represented by the name. The value returned by the assignment expression is the unit value.

Properties

The maximal type of an assignment_expr is **Unit**.

An assignment_expr writes to the variable represented by the constituent name and accesses any of the variables or channels which the constituent expr accesses.

Context conditions

The name must represent a variable.

The maximal type of the expr must be less than or equal to the maximal type of the name.

7.21 Input expressions

Syntax

input_expr ::=

channel-name ?

Meaning

The effect of an input expression is to offer an input communication from the channel represented by the name. The value returned by the input expression is the value input from the channel.

Properties

The maximal type of an input_expr is the maximal type of the constituent name.

An input_expr inputs from the channel represented by the constituent name.

Context conditions

The name must represent a channel.

7.22 Output expressions

Syntax

output_expr ::=
channel-name ! expr

Meaning

The effect of an output expression is to offer an output communication to the channel represented by the name, of the value returned by the constituent expression. The value returned by the output expression is the unit value.

Properties

The maximal type of an output_expr is **Unit**.

An output_expr outputs to the channel represented by the constituent name and accesses any of the variables or channels which the constituent expr accesses.

Context conditions

The name must represent a channel.

The maximal type of the expr must be less than or equal to the maximal type of the name.

7.23 Structured expressions

Syntax

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

7.23.1 Local expressions

Syntax

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

Meaning

The effect of a local expression is the effect of the constituent expression evaluated in the scope of the declarations. The names defined by the declarations may be under-specified thus resulting in a set of environments. A non-deterministic choice is made between the effects of evaluating the constituent expression in these environments. The local expression is thus capable of introducing non-determinism.

If the constituent expression offers a communication via a locally declared channel to the outside world, the local expression deadlocks.

Properties

The scope of the opt-decl-string is opt-decl-string itself and the expr. Note, that this means that the order of the definitions in the opt-decl_string is indifferent.

The maximal type of a local_expr is the maximal type of the expr.

A local_expr accesses any of the non-local variables and channels (i.e. variables and channels not defined in the opt-decl-string) which the expr accesses.

7.23.2 Let expressions

Syntax

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

let_def ::=
  typing |
  explicit_let |
  implicit_let

explicit_let ::=
  let_binding = expr

implicit_let ::=
  single_typing restriction

let_binding ::=
  binding |
  record_pattern |
  list_pattern

Meaning

The effect of a let expression is the effect of the constituent expression evaluated in the scope of the definitions occurring in the let definition list. A let expression – with only a single let definition – of the form

```
let let_def in
  expr
end
```

defines through the let-definition local names to be visible only within the constituent expression. The names defined by the let-definition may be under-specified thus resulting in a set of environments. A non-deterministic choice is made between the effects of evaluating the constituent expression in these environments. The let expression is thus capable of introducing non-determinism.

There are three kinds of let-definitions.

- A let-definition of the form of a *typing* represents the set of environments represented by the typing.
- A let-definition of the form of an *implicit_let* represents a subset of the environments that the single typing represents: those that makes the restriction hold.
- A let-definition of the form of an *explicit_let* represents the set of environments obtained as follows. The expression is evaluated to return a value which is then matched against the let-binding resulting in a set of environments.

A let expression involving more than one let definition is a shorthand for a number of nested let expressions with single let definitions. That is, a let expression of the form

let let_def$_1$, ... ,let_def$_n$ **in**
  expr
end

is a shorthand for

let let_def$_1$ **in**
  $\vdots$
    let let_def$_n$ **in**
      expr
    end
  $\vdots$
end

Properties

In a let_expr of the form

let let_def$_1$, ... ,let_def$_n$ **in**
  expr
end

the scope of let_def$_i$ ($1 \leq i \leq n$) is expr and all let_def$_j$ for $j > i$.

The maximal type of a let_expr is the maximal type of the expr.

In an explicit_let the maximal context type of the let_binding is the maximal type of the expr.

A let_expr accesses any of the variables and channels which the constituent let_defs and expr access.

A typing does not access any variables or channels.

An explicit_let accesses any of the variables and channels which the constituent expr accesses.

An implicit_let reads any of the variables which the constituent restriction reads.

7.23.3 If expressions

Syntax

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

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

else_branch ::=
  else expr

Meaning

The effect of an if expression is to determine the applicable alternative followed by the effect of that alternative. An if expression of the form

if expr_1 **then** expr_2 **else** expr_3 **end**

is evaluated by evaluating the first constituent expression to return a boolean value – the test value. If the test value is equal to **true**, the second constituent expression is evaluated. Alternatively, if the test value is equal to **false** the third constituent expression is evaluated.

An if expression involving elsif-branches is a shorthand for a number of nested if expressions without elsif-branches. An if expression of the form

$$
\begin{array}{l}
\textbf{if}\ \mathrm{expr}_1\ \textbf{then}\ \mathrm{expr}_1{}'\\
\textbf{elsif}\ \mathrm{expr}_2\ \textbf{then}\ \mathrm{expr}_2{}'\\
\vdots\\
\textbf{elsif}\ \mathrm{expr}_n\ \textbf{then}\ \mathrm{expr}_n{}'\\
\mathrm{opt\_else\_branch}\\
\textbf{end}
\end{array}
$$

is a shorthand for

$$
\begin{array}{l}
\textbf{if}\ \mathrm{expr}_1\ \textbf{then}\ \mathrm{expr}_1{}'\ \textbf{else}\\
\quad\textbf{if}\ \mathrm{expr}_2\ \textbf{then}\ \mathrm{expr}_2{}'\ \textbf{else}\\
\qquad\vdots\\
\qquad\quad\textbf{if}\ \mathrm{expr}_n\ \textbf{then}\ \mathrm{expr}_n{}'\ \mathrm{opt\_else\_branch}\ \textbf{end}\\
\qquad\vdots\\
\quad\textbf{end}\\
\textbf{end}
\end{array}
$$

An if expression of the form

$$\textbf{if}\ \mathrm{expr}_1\ \textbf{then}\ \mathrm{expr}_2\ \textbf{end}$$

is a shorthand for

$$\textbf{if}\ \mathrm{expr}_1\ \textbf{then}\ \mathrm{expr}_2\ \textbf{else}\ \textbf{skip}\ \textbf{end}$$

Properties

The maximal type of an if_expr is the least upper bound of the maximal type of the second constituent expr and all the constituent branches.

The maximal type of an else_if_branch is the maximal type of second constituent expr.

The maximal type of an else_branch is the maximal type of the constituent expr.

An if_expr accesses any of the variables and channels which the constituent exprs and branches access.

An elsif_branch accesses any of the variables and channels which the constituent exprs access.

An else_branch accesses any of the variables and channels which the constituent expr accesses.

Context conditions

In an if_expr the first expr must have the maximal type **Bool**. The maximal types of the second expr and all the constituent branches must have a least upper bound.

In an else_if_branch the first expr must have the maximal type **Bool**.

7.23.4 Case expressions

Syntax

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

case_branch ::=
  pattern $\rightarrow$ expr

Meaning

The effect of a case expression is to evaluate the constituent expression, determine the matching case branch and then to evaluate the expression part of that case branch.

The constituent expression is evaluated to return a value – the test value. Then the case branches are processed from left to right until the test value succeedes to match a pattern. The successful pattern matching then results in a set of environments. The corresponding expression in the matching case branch is then evaluated in each of these environments and a non-deterministic choice is made between the resulting effects.

If there is no matching case branch, the whole case expression defaults.

Properties

In a case_branch the scope of the pattern is the expr.

The maximal type of a case_expr is the least upper bound of the maximal types of the exprs in the constituent case_branches.

In a case_expr the maximal context type of the patterns in the case_branches is the maximal type of the expr.

A case_expr accesses any of the variables and channels which the constituent expr and case_branches access.

A case_branch accesses any of the variables and channels which the constituent expr accesses.

Context conditions

In a case_expr the maximal types of the exprs in the constituent case_branches must have a least upper bound.

7.23.5 For expressions

Syntax

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

Meaning

The effect of a for expression is to repeat the evaluation of the constituent expression for each element of a list value. For each environment in the list of environments represented by the list limitation – processed from left to right – the constituent expression is evaluated. The value returned by the for expression is the unit value.

Properties

In a for_expr the scope of the list_limitation extends to the expr.

The maximal type of a for_expr is **Unit**.

A for_expr accesses any of the variables and channels which the constituent list_limitation and expr access.

Context conditions

The expr must have the maximal type **Unit**.

7.23.6 While expressions

Syntax

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

Meaning

The effect of a while expression is to repeat the evaluation of the second constituent expression while the first boolean constituent expression evaluates to **true**. The value returned by the while expression is the unit value.

A while expression of the form

  while expr_1 **do** expr_2 **end**

is equivalent to

  if expr_1 **then**
    expr_2 ; **while** expr_1 **do** expr_2 **end**
  else
    skip
  end

Properties

The maximal type of a while_expr is **Unit**.

A while_expr accesses any of the variables and channels which the constituent exprs access.

Context conditions

The first expr must have the maximal type **Bool**. The second expr must have the maximal type **Unit**.

7.23.7 Until expressions

Syntax

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

Meaning

The effect of an until expression is to repeat the evaluation of the first constituent expression until the second boolean constituent expression evaluates to **true**. The value returned by the until expression is the unit value.

An until expression of the form

  do expr_1 **until** expr_2 **end**

is a shorthand for

  expr_1 ; **while** $\sim\mathrm{expr}_2$ **do** expr_1 **end**

Properties

The maximal type of an until_expr is **Unit**.

An until_expr accesses any of the variables and channels which the constituent exprs access.

Context conditions

The first expr must have the maximal type **Unit**. The second expr must have the maximal type **Bool**.

7.24 Expression lists

Meaning

An expression list is evaluated either as a product or as a list.

- The effect of an *expression list evaluated as a product* is to return a value obtained as follows.

  The value of an expression list containing a single expression:

  expr

  is the value returned by the expression. The value of an expression list containing more than one expression:

  $\mathrm{expr}_1, \ldots, \mathrm{expr}_n$

  is obtained by evaluating each expression $expr_i$ to yield a resulting value v_i and then forming the product value:

  $(\mathrm{v}_1, \ldots, \mathrm{v}_n)$

  An optional expression list is evaluated as follows. If the expression list is absent, the resulting value is the unit value '()' of type **Unit**. If the expression list is present, the resulting value is the resulting value of the expression list evaluated as a product.

- The effect of an *expression list evaluated as a list* is to return a list obtained as follows.

  The resulting value of an expression list containing a single expression:

  expr

  is the one-element list containing the resulting value of the expression. The value of an expression list containing more than one expression:

  $\mathrm{expr}_1, \ldots, \mathrm{expr}_n$

  is obtained by evaluating each expression $expr_i$ to yield a resulting value v_i and then forming the list value:

  $\langle \mathrm{v}_1, \ldots, \mathrm{v}_n \rangle$

  An optional expression list is evaluated as follows. If the expression list is absent, the resulting value is the empty list. If the expression list is present, the resulting value is the resulting value of the expression list evaluated as a list.

8 Bindings

Syntax

binding ::=
  id_or_op |
  product_binding

product_binding ::=
  (binding-list2)

Terminology

An *environment* is a mapping from identifiers and operators to values. One environment env_1 can be overwritten with another environent env_2 using †. That is, the environment resulting from

$$\mathrm{env}_1 \dagger \mathrm{env}_2$$

is equal to env_2 for all the identifiers and operators for which env_2 is defined. For identifiers and operators only defined by env_1 the resulting environment equals env_1.

A *decomposer* is a mapping from values to environments. *Matching* a value against a decomposer means to apply the decomposer to the value and thus obtain an environment.

Meaning

A binding represents a decomposer. In the below explanation we shall use the convention of writing

$$\mathrm{b(v)}$$

for the environment obtained by matching the value v against the (decomposer represented by the) binding b.

The environment obtained by matching the value v against a binding is defined as follows. In case the binding is

- an *id_or_op* then the environment obtained is

[id_or_op $\mapsto$ v]

- a *product_binding* of the form

  (binding$_1$, ... ,binding$_n$)

  then v must be a product value of the form

  (v$_1$, ... ,v$_n$)

  and the resulting environment is

  binding$_1$(v$_1$) † ... † binding$_n$(v$_n$)

Properties

The context of a binding determines a *maximal context type* for the binding. For constructs containing bindings, this maximal context type is stated.

An id_or_op being a binding has as maximal type the maximal context type of the binding.

In a product_binding of the form (b$_1$, ..., b$_n$) having a context type of the form t$_1$ × ... × t$_n$, the maximal context types of the constituent bindings b$_1$, ..., b$_n$ are t$_1$, ..., t$_n$, respectively.

Context conditions

The maximal context type of a product_binding must be a product type of the same length as the binding-list2. The names introduced in the constituent bindings must be distinct unless they have distinguishable maximal types.

9 Typings

Syntax

typing ::=
  single_typing |
  multiple_typing

single_typing ::=
  binding : type_expr

multiple_typing ::=
  binding-list2 : type_expr

Meaning

The basic form of typing is the single typing. All multiple typings and typing lists are shorthands for single typings. A single typing represents

- a type t – represented by the type expression.
- a decomposer d – represented by the binding – against which a value can be matched to yield an environment.
- a set of environments – obtained by applying the decomposer to each value in the type:

  $\{d(v) \mid v \in t\}$

A multiple typing of the form

  binding$_1$, ... ,binding$_n$: type_expr

is a shorthand for the single typing

  (binding$_1$, ... ,binding$_n$) : type_expr × ... × type_expr

where the product type expression has length n.

A typing list of the form

binding$_1$: type_expr$_1$, ... ,binding$_n$: type_expr$_n$

is a shorthand for a single typing

(binding$_1$, ... ,binding$_n$) : type_expr$_1$ $\times$ ... $\times$ type_expr$_n$

A typing list involving multiple typings is first re-written by re-writing the multiple typings into single typings.

An optional typing list where the typing list is present, represents the type and decomposer represented by the typing list. In case the typing list is absent, the type and decomposer are as follows:

- the type is **Unit**.
- the decomposer is the wildcard decomposer which when matched against a value yields the empty environment.

Properties

The maximal type of a single_typing is the maximal type of the type_expr.

The maximal type of a multiple_typing is the maximal type of the single_typing it is a shorthand for.

In a typing the maximal context type of the constituent bindings is the maximal type of the constituent type_expr.

Context conditions

In a multiple_typing the names introduced in the constituent bindings must be distinct unless they have distinguishable maximal types.

10 Patterns

Syntax

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

Terminology

A pattern has two roles:

- To control, in conditional contexts, the choice between alternatives on the basis of pattern matching.
- To provide names for the constituent parts of compound values.

Matching a value against a pattern yields either *failure* or *success*. In the case of success the result of the matching is a set of environments, each mapping identifiers and operators occurring in the pattern into constituent parts of the value.

The fact that the result of a successful pattern matching is a set (of environments) is due to record patterns that may introduce non-deterministic decomposition of compound values.

In the following explanation of patterns we shall use the convention of writing

  p(v)

for the set of environments obtained by the successful matching of the value v against the pattern p.

For each pattern kind, the criteria for match success is given together with the resulting environments in case of match success. The value matched against the pattern will be referred to as the test value.

Properties

The context of a pattern determines a *maximal context type* for the pattern. For constructs containing patterns, this maximal context type is stated.

10.1 Value literals

Meaning

- *match success*: The value literal must be equal to the test value.
- *resulting environment set*: {[ ]}

Context conditions

The maximal type of a value_literal considered as an expr must be less than or equal to the maximal context type of the value_literal.

10.2 Names

Meaning

- *match success*: The value represented by the name must be equal to the test value.
- *resulting environment set*: {[ ]}

Context conditions

For a pattern being a name, the name must represent a value.

The maximal type of the name must be less than or equal to the maximal context type of the pattern.

10.3 Wildcard patterns

Syntax

wildcard_pattern ::=

_

Meaning

- *match success*: All values match a wildcard pattern.
- *resulting environment set*: {[ ]}

10.4 Product patterns

Syntax

product_pattern ::=
  (pattern-list2)

Meaning

- *match success*: If the product pattern is of the form

  $(\text{pattern}_1, \ldots ,\text{pattern}_n)$

  then the test value must be a product value of the form

  $(\text{v}_1, \ldots ,\text{v}_n)$

  and each v_i must additionally match the corresponding pattern $pattern_i$.

- *resulting environment set*:

  $\{\text{environment}_1 \dagger \ldots \dagger \text{environment}_n \mid$
  $\quad \text{environment}_1 \in \text{pattern}_1(\text{v}_1)$
  $\quad \wedge \ldots \wedge$
  $\quad \text{environment}_n \in \text{pattern}_n(\text{v}_n)\}$

Properties

In a product_pattern of the form $(\text{pattern}_1, \ldots, \text{pattern}_n)$ having a maximal context type of the form $\text{t}_1 \times \ldots \times \text{t}_n$, the maximal context type of the constituent patterns $\text{pattern}_1, \ldots, \text{pattern}_n$ is $\text{t}_1, \ldots, \text{t}_n$, respectively.

Context conditions

The maximal context type of a product_pattern must be a product type of the same length as the pattern-list2.

The names introduced in the constituent patterns must be distinct unless they have distinguishable maximal types.

10.5 Record patterns

Syntax

record_pattern ::=
  pure_value-name component_patterns

component_patterns ::=
  (inner_pattern-list)

inner_pattern ::=
  binding |
  wildcard_pattern

Meaning

- *match success*: The name must represent a function c:

  $$\mathrm{c} : \mathrm{t_0} \xrightarrow{\sim} \mathrm{t}$$

  where t is the type of the test value. Let v be the test value, then there must exist at least one value $v_0 : t_0$ such that:

  $$\mathrm{c(v_0) = v}$$

  By this is meant that c when applied to v_0 is converging and deterministically returning the value v.

- *resulting environment set*: Let cp be the component pattern, the meaning of which is described below. The resulting set of environments is thus:

  $$\{\ \mathrm{cp(v_0)} \mid \mathrm{v_0} \in \mathrm{t_0} \wedge \mathrm{c(v_0) = v}\ \}$$

Component patterns are deterministic since they cannot involve record patterns. When a value is matched against a component pattern, the result is thus a single environment.

The environment resulting from matching a value against a component pattern of the form

(inner_pattern)

is the environment obtained by matching the value against the *inner_pattern*.

The environment resulting from matching a value of the form

(v_1, ... , v_n)

against a component pattern of the form

(inner_pattern$_1$, ... ,inner_pattern$_n$)

is

inner_pattern$_1$(v_1) † ... † inner_pattern$_n$(v_n)

The environment resulting from matching a value against an inner pattern with the form of a binding is the environment obtained by matching the value against the binding.

The environment resulting from matching a value against an inner pattern with the form of a wildcard pattern is the empty environment.

Properties

In a record_pattern the maximal context type of the component_patterns is the domain part of the maximal type of the name.

In a component_pattern of the form (p_1, ..., p_n) having a maximal context type of the form $t_1 \times ... \times t_n$, the maximal context types of the constituent inner_patterns p_1, ..., p_n is t_1, ..., t_n, respectively.

In a component_pattern of the form (p) having a context type t the maximal context type of the constituent inner_pattern p is t.

Context conditions

In a record_pattern the name must represent a value and have a maximal type which is a function type. The result type part of this type must be less than or equal to the maximal context type of the record_pattern.

In a component_patterns the names introduced in the constituent inner_patternss must be distinct unless they have distinguishable maximal types. A component_pattern of the form (p_1, ..., p_n) , n>1, must have a maximal context type of the form $t_1 \times ... \times t_n$.

10.6 List patterns

Syntax

list_pattern ::=
  constructed_list_pattern |
  left_list_pattern |
  right_list_pattern |
  left_right_list_pattern

Context conditions

The maximal context type of a list_pattern must be a list type.

10.6.1 Constructed list patterns

Syntax

constructed_list_pattern ::=
  ⟨ opt-inner_pattern-list ⟩

Meaning

- *match success*: When the constructed list pattern is of the form

  $\langle \text{inner\_pattern}_1, \ldots ,\text{inner\_pattern}_n \rangle$

  the test value must be a list of the form

$\langle \mathrm{v}_1, \ldots, \mathrm{v}_n \rangle$

We shall refer to n as the length of the constructed list pattern. The criteria for match success can thus be re-formulated as "the length of the test value must be equal to the length of the constructed list pattern".

- *resulting environment set*:

  $\{\text{inner\_pattern}_1(\mathrm{v}_1) \dagger \ldots \dagger \text{inner\_pattern}_n(\mathrm{v}_n)\}$

Properties

The maximal context type of each of the constituent inner_patterns is the element part of the maximal context type of the constructed_list_pattern.

Context conditions

The names introduced in the constituent inner_patternss must be distinct unless they have distinguishable maximal types.

10.6.2 Left list patterns

Syntax

left_list_pattern ::=
  constructed_list_pattern ⌢ id_or_wildcard

id_or_wildcard ::=
  id |
  wildcard_pattern

Meaning

For an inner pattern we have

- *match success*: The test value must be a list and its length must be greater than or equal to the length of the constructed list pattern. Thus if the left list pattern is of the form

  $\langle \text{inner\_pattern}_1, \ldots, \text{inner\_pattern}_n \rangle \frown \text{id\_or\_wildcard}$

then the test value must be a list of the form

$\langle v_1, \ldots, v_n \rangle \frown$ suffix

- *resulting environment set*:

  {inner_pattern$_1$(v$_1$) † ... † inner_pattern$_n$(v$_n$)
  † id_or_wildcard(suffix)}

For an id or wild card pattern we have

- *match success*: All values match an id or wildcard pattern.
- *resulting environment set*:

  Matching a value v against an id_or_wildcard pattern of the form of an id yields the environment set

  { [id $\mapsto$ v] }

  and of the form of a wildcard yields the environment set{[ ]}

Properties

In a left_list_pattern the maximal context type of the constituent constructed_list_pattern and id_or_wildcard is the maximal context type of the left_list_pattern.

The maximal type of an id in an id_or_wildcard is the maximal context type.

Context conditions

The names introduced in the constructed_list_pattern and in id_or_wildcard must be distinct unless they have distinguishable maximal types.

10.6.3 Right list patterns

Syntax

right_list_pattern ::=
  id_or_wildcard $\frown$ constructed_list_pattern

Meaning

- *match success*: The test value must be a finite list and its length must be greater than or equal to the length of the constructed list pattern. Thus if the right list pattern is of the form

  id_or_wildcard $\frown$ $\langle$inner_pattern$_1$, ... ,inner_pattern$_n\rangle$

  then the test value must be of the form

  prefix $\frown$ $\langle$v$_1$, ... ,v$_n\rangle$

- *resulting environment set*:

  {id_or_wildcard(prefix) †
  inner_pattern$_1$(v$_1$) † ... † inner_pattern$_n$(v$_n$)}

Properties

In a right_list_pattern the maximal context type of the constituent constructed_list_pattern and id_or_wildcard is the maximal context type of the right_list_pattern itself.

Context conditions

The names introduced in the constructed_list_pattern and in id_or_wildcard must be distinct unless they have distinguishable maximal types.

10.6.4 Left right list patterns

Syntax

left_right_list_pattern ::=
  constructed_list_pattern $\frown$ id_or_wildcard $\frown$ constructed_list_pattern

Meaning

- *match success*: The test value must be a finite list and its length must be greater than or equal to the sum of the lengths of the two constructed list patterns. Thus if the left right list pattern is of the form

$\langle$inner_pattern$_{1,1}$, ... ,inner_pattern$_{1,n_1}\rangle$
$\frown$ id_or_wildcard $\frown$
$\langle$inner_pattern$_{2,1}$, ... ,inner_pattern$_{2,n_2}\rangle$

then the test value must be of the form

$\langle$v$_{1,1}$, ... ,v$_{1,n_1}\rangle$
$\frown$ infix $\frown$
$\langle$v$_{2,1}$, ... ,v$_{2,n_2}\rangle$

- *resulting environment set*:

{
inner_pattern$_{1,1}$(v$_{1,1}$) † ... † inner_pattern$_{1,n_1}$(v$_{1,n_1}$)
† id_or_wildcard(infix) †
inner_pattern$_{2,1}$(v$_{2,1}$) † ... † inner_pattern$_{2,n_2}$(v$_{2,n_2}$)
}

Properties

In a left_right_list_pattern the maximal context type of the constituent constructed_list_patterns and id_or_wildcard is the maximal context type of the left_right_list_pattern itself.

Context conditions

The names introduced in the constructed_list_patterns and in id_or_wildcard must be distinct unless they have distinguishable maximal types.

11 Names

Syntax

name ::=
  qualified_id |
  qualified_op

Meaning

A name represents an entity such as a scheme, object, type, value, variable or channel.

Properties

If a name represents a value, a variable, a channel or a type then it has an associated maximal type.

11.1 Qualified identifiers

Syntax

qualified_id ::=
  opt-qualification id

qualification ::=
  element-object_expr .

Meaning

An un-qualified identifier represents the entity to which it has been bound by an enclosing definition.

A qualified identifier represents the entity obtained by looking up the identifier in the model represented by the qualification.

A qualification represents the model represented by the object expression.

Properties

A qualified_id represents the entity represented by the constituent id.

The maximal type of a qualified_id representing a value, a variable, a channel or a type is the maximal type of the constituent id.

In a qualified_id the scope of a qualification is extended to the id, while all other definitions are hidden there.

Context conditions

In a qualification the object_expr must represent a model.

11.2 Qualified operators

Syntax

qualified_op ::=
  opt-qualification (op)

Meaning

An un-qualified operator in brackets represents a function value. The operator can either be predefined or it can have been introduced in an enclosing definition. If the operator has been introduced by a definition, the choice between predefined and defined version depends on overload-resolution.

A qualified operator represents the function obtained by looking up the operator in the model represented by the qualification.

The brackets turn the operator into a function that must be applied with prefix notation via an application expression. Assume the prefix operator *p_op* and the infix binary operator *i_op*, then the following equivalences hold:

$$\mathrm{p\_op\ expr} \equiv \mathrm{(p\_op)(expr)}$$

$$\mathrm{expr}_1\ \mathrm{i\_op}\ \mathrm{expr}_2 \equiv \mathrm{(i\_op)(expr}_1\mathrm{,expr}_2\mathrm{)}$$

Properties

A qualified_op represents the value represented by the constituent op.

The maximal type of a qualified_op is the maximal type of the constituent op.

In a qualified_op in which a qualification is present the scope of this qualification is extended to the op, while all other definitions are hidden there.

In a qualified_op in which no qualification is present all predefined polymorphic meanings of operators are hidden.

11.3 Identifiers and operators

Syntax

id_or_op ::=
  id |
  op

op ::=
  infix_op |
  prefix_op

Properties

Each occurrence of an identifier or operator (id, op or id_or_op) is either a *defining* or an *applied* occurrence.

The following occurrences are defining occurrences:

- The id (or ids) occurring immediately within a scheme_def, object_def, axiom_naming, variable_def, channel_def, sort_def variant_def, union_def, short_record_def, abbreviation_def, subtype_naming, prefix_application, infix_application and id_or_wildcard.
- The new id_or_op in a rename_pair.
- The id_or_op occurring immediately within a constructor, destructor, reconstructor and binding.

All other occurrences are applied occurrences.

Each defining occurrence of an identifier or operator is part of a declarative construct that represents at least a definition introducing this identifier or operator.

An applied occurrence of an identifier or operator is said to be *visible* if there is a visible definition introducing it.

A legal applied occurrence of an identifier or operator has a *corresponding definition* (or *interpretation*). There are three cases for an applied occurrence of an identifier or operator:

1. There is no visible definition introducing it, i.e. it is not visible. In that case the occurrence is illegal, cf. the context condition below, and hence the identifier or operator has no corresponding definition.
2. There is exactly one visible definition introducing it. This definition is the corresponding definition of the identifier or operator.
3. There are two or more visible definitions introducing it. According to the visibility rules and context conditions for declarative constructs this can only be the case for names of values. In the section on overloading it is explained how to find the corresponding definition in this case, if possible.

An applied occurrence of an identifier or operator *represents* the entity of its corresponding definition.

For an identifier or operator representing a value, a variable, a channel or a type, its maximal type is determined by its corresponding definition.

Note, that all operators have one or more predefined meanings which has the whole specification as scope. A predefined meaning is said to be *polymorphic* if its type contains type variables.

Context conditions

An applied occurrence of an identifier and operator must be visible.

11.3.1 Infix operators

Syntax

infix_op ::=
  $=$ |
  $\neq$ |
  $>$ |

$<$ |
$\geq$ |
$\leq$ |
$\supset$ |
$\subset$ |
$\supseteq$ |
$\subseteq$ |
$\in$ |
$\notin$ |
$+$ |
$-$ |
$\backslash$ |
$\hat{}$ |
$\cup$ |
$\dagger$ |
$*$ |
$/$ |
$\circ$ |
$\cap$ |
$\uparrow$ |
$

Meaning

Below, the predefined meanings of the infix operators are stated.

The infix operators operate on pairs of values referred to as arguments. Some operators may have pre-conditions that must hold for the arguments. When a pre-condition is violated the result of the value infix expression is not well-defined.

The type T and subscripted versions of T occurring in the operator signatures are type variables representing arbitrary types.

- **Equal:**

  $= : \mathrm{T} \times \mathrm{T} \rightarrow \mathbf{Bool}$

  The result is **true** iff. the two arguments are equal.

- **Not equal:**

  $\neq : \mathrm{T} \times \mathrm{T} \rightarrow \mathbf{Bool}$

  The result is **true** iff. the two arguments are not equal.

- **Integer addition:**

  $+ : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$

  The result is the sum of the two integers.

- **Real addition:**

  $+ : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$

  The result is the sum of the two reals.

- **Integer subtraction:**

  $- : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$

  The result is the difference between the first integer and the second integer.

- **Real subtraction:**

  $- : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$

  The result is the difference between the first real and the second real.

- **Integer multiplication:**

  $* : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$

  The result is the product of the two integers.

- **Real multiplication:**

  $* : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$

  The result is the product of the two reals.

- **Integer exponentiation:**

  $\uparrow : \mathbf{Int} \times \mathbf{Int} \stackrel{\sim}{\to} \mathbf{Real}$

  Pre-condition: If the second integer is negative the first integer must be different from zero (0).

  The result is the first integer raised to the power of the second integer.

- **Real exponentiation:**

  $\uparrow : \mathbf{Real} \times \mathbf{Real} \stackrel{\sim}{\to} \mathbf{Real}$

Pre-condition: If the second real is negative the first real must be different from zero (0). If the second real is not a whole number the first real must be non-negative.

The result is the first real raised to the power of the second real.

- **Function composition:**

  $\circ : (\mathrm{T}_2 \stackrel{\sim}{\rightarrow} \mathrm{acc}\ \mathrm{T}_3) \times (\mathrm{T}_1 \stackrel{\sim}{\rightarrow} \mathrm{acc}'\ \mathrm{T}_2) \rightarrow (\mathrm{T}_1 \stackrel{\sim}{\rightarrow} \mathrm{acc}''\mathrm{T}_3)$

  where acc" is the union of acc and acc'.

  The result is the composition of the two functions defined as follows:

  $(\mathrm{expr}_1 \circ \mathrm{expr}_2)(\mathrm{expr}) \equiv \mathrm{expr}_1(\mathrm{expr}_2(\mathrm{expr}))$

- **Map composition:**

  $\circ : (\mathrm{T}_2 \underset{m}{\rightarrow} \mathrm{T}_3) \times (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2) \rightarrow (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_3)$

  The result is the composition of the two maps defined as follows:

  $(\mathrm{expr}_1 \circ \mathrm{expr}_2)(\mathrm{expr}) \equiv \mathrm{expr}_1(\mathrm{expr}_2(\mathrm{expr}))$

- **Integer division:**

  $/ : \mathbf{Int} \times \mathbf{Int} \stackrel{\sim}{\rightarrow} \mathbf{Int}$

  Pre-condition: The second integer must not be zero (0).

  The absolute value (without sign) of the result is the number of times that the absolute value of the second integer can be within the absolute value of the first integer. The sign of the result is the traditional product of the signs of the arguments.

- **Real division:**

  $/ : \mathbf{Real} \times \mathbf{Real} \stackrel{\sim}{\rightarrow} \mathbf{Real}$

  Pre-condition: The second real must not be zero (0).

  The result is obtained by dividing the first real with the second real.

- **Map restriction to:**

  $/ : (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2) \times \mathrm{T}_1\text{-}\mathbf{infset} \rightarrow (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2)$

  The result is the map with its domain limited to the elements of the set.

- **Integer remainder:**

$\backslash$: **Int** $\times$ **Int** $\xrightarrow{\sim}$ **Int**

Pre-condition: The second integer must not be zero (0).

The absolute value of the result is the remainder after having divided the absolute value of the second integer into the absolute value of the first integer. The sign of the result is the sign of the first integer. This implies the following relation between integer division and integer remainder. Let a and b be integers:

a = (a/b)*b + (a\b)

- **Set difference:**

  $\backslash$: T-**infset** $\times$ T-**infset** $\rightarrow$ T-**infset**

  The result is the set of all elements which appear in the first set and not in the second.

- **Map restriction with:**

  $\backslash$: $(\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2) \times \mathrm{T}_1$-**infset** $\rightarrow (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2)$

  The result is the map with the elements of the set removed from its domain.

- **Integer greater than:**

  $>$: **Int** $\times$ **Int** $\rightarrow$ **Bool**

  The result is **true** iff. the first integer is greater than the second integer.

- **Real greater than:**

  $>$: **Real** $\times$ **Real** $\rightarrow$ **Bool**

  The result is **true** iff. the first real is greater than the second real.

- **Integer less than:**

  $<$: **Int** $\times$ **Int** $\rightarrow$ **Bool**

  The result is **true** iff. the first integer is less than the second integer.

- **Real less than:**

  $<$: **Real** $\times$ **Real** $\rightarrow$ **Bool**

  The result is **true** iff. the first real is less than the second real.

- **Integer greater than or equal to:**

$\geq$: **Int** $\times$ **Int** $\rightarrow$ **Bool**

The result is **true** iff. the first integer is greater than or equal to the second integer.

- **Real greater than or equal to:**

  $\geq$: **Real** $\times$ **Real** $\rightarrow$ **Bool**

  The result is **true** iff. the first real is greater than or equal to the second real.

- **Integer less than or equal to:**

  $\leq$: **Int** $\times$ **Int** $\rightarrow$ **Bool**

  The result is **true** iff. the first integer is less than or equal to the second integer.

- **Real less than or equal to:**

  $\leq$: **Real** $\times$ **Real** $\rightarrow$ **Bool**

  The result is **true** iff. the first real is less than or equal to the second real.

- **Proper superset:**

  $\supset$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  The result is **true** iff. the second set is a proper subset of the first set. That is, it is a subset of the first set but not equal to it.

- **Proper subset:**

  $\subset$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  The result is **true** iff. the first set is a proper subset of the second set. That is, it is a subset of the second set but not equal to it.

- **Superset:**

  $\supseteq$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  The result is **true** iff. the second set is a subset of the first set.

- **Subset:**

  $\subseteq$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  The result is **true** iff. the first set is a subset of the second set.

- **Within:**

  $\in : \mathrm{T} \times \mathrm{T}\text{-}\mathbf{infset} \rightarrow \mathbf{Bool}$

  The result is **true** iff. the first argument is a member of the set.

- **Not within:**

  $\notin : \mathrm{T} \times \mathrm{T}\text{-}\mathbf{infset} \rightarrow \mathbf{Bool}$

  The result is **true** iff. the first argument is not a member of the set.

- **Intersection:**

  $\cap : \mathrm{T}\text{-}\mathbf{infset} \times \mathrm{T}\text{-}\mathbf{infset} \rightarrow \mathrm{T}\text{-}\mathbf{infset}$

  The result is the set containing all elements which appear in both of the two sets.

- **Set union:**

  $\cup : \mathrm{T}\text{-}\mathbf{infset} \times \mathrm{T}\text{-}\mathbf{infset} \rightarrow \mathrm{T}\text{-}\mathbf{infset}$

  The result is the set containing all elements which appear in one or both of the two sets.

- **Map union:**

  $\cup : (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2) \times (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2) \rightarrow (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2)$

  The result is the map containing all the pairs of the first map and all the pairs of the second map. Note that if the intersection of the domains of the two maps is not empty, the union may lead to a non-deterministic map.

- **List concatenation:**

  $\frown : \mathrm{T}^* \times \mathrm{T}^\omega \xrightarrow{\sim} \mathrm{T}^\omega$

  The result is the concatenation of the two lists. That is, the list containing all the elements of the two lists, ordered as in the two lists and with all the elements of the first list appearing first.

- **Map overwrite:**

  $\dagger : (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2) \times (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2) \rightarrow (\mathrm{T}_1 \underset{m}{\rightarrow} \mathrm{T}_2)$

  The result is the first map overwritten with the second map. Where the two maps have common domain elements, the second map overwrites the first.

- **Set distribution:**

$\$: (T \times T \xrightarrow{\sim} T) \times \textbf{T-set} \xrightarrow{\sim} T$

Pre-condition: The function, say f, must be associative, be commutative and have exactly one unit u such that:

$\forall\ x : T \bullet f(u,x) = x$

The result is the value obtained by applying the function to 'all the elements' of the set – pair-wise:

$$
\begin{array}{l}
f\ \$\ s \equiv \\
\quad \textbf{if}\ s = \{\}\ \textbf{then} \\
\quad\quad u \\
\quad \textbf{else} \\
\quad\quad \textbf{let}\ x : T \bullet x \in s\ \textbf{in} \\
\quad\quad\quad f(x,f\ \$\ s\backslash\{x\}) \\
\quad\quad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

The above definition of f applied to a set with more than one element depends on a repeated non-deterministic choice of element from the set. This does, however, not make the result non-deterministic since f is commutative and associative.

- **List distribution:**

$\$: (T \times T \xrightarrow{\sim} T) \times T^{*} \xrightarrow{\sim} T$

Pre-condition: The function, say f, must be associative and have exactly one right unit u such that:

$\forall\ x : T \bullet f(x,u) = x$

The result is the value obtained by applying the function to 'all the elements' of the list – pair-wise – in order left to right:

$$
\begin{array}{l}
f\ \$\ l \equiv \\
\quad \textbf{if}\ l = \langle\rangle\ \textbf{then} \\
\quad\quad u \\
\quad \textbf{else} \\
\quad\quad f(\textbf{hd}\ l,f\ \$\ (\textbf{tl}\ l)) \\
\quad \textbf{end}
\end{array}
$$

11.3.2 Prefix operators

Syntax

prefix_op ::=
  abs |
  it |
  rl |
  card |
  len |
  inds |
  elems |
  hd |
  tl |
  front |
  last |
  dom |
  rng

Meaning

Below, the predefined meanings of the prefix operators are stated.

The prefix operators operate on values (arguments). Some operators may have pre-conditions that must hold for the argument. When a pre-condition is violated the result of the value prefix expression is not well-defined.

The type T occurring in the operator signatures is a type variable representing an arbitrary type.

- **Absolute value of integer:**

  abs : **Int** $\rightarrow$ **Nat**

  The result is the absolute value of the integer. That is, if the integer is negative, the negated value is returned. The operator is the identity on non-negative integers.

- **Absolute value of real:**

  abs : **Real** $\rightarrow$ **Real**

  The result is the absolute value of the real. That is, if the real is negative, the negated value is returned. The operator is the identity on non-negative reals.

- **Real to integer conversion:**

  $\textbf{it} : \textbf{Real} \to \textbf{Int}$

  The absolute value (without sign) of the result is the greatest integer that is smaller than or equal to the absolute value of the real. the sign is the sign of the real.

- **Integer to real conversion:**

  $\textbf{rl} : \textbf{Int} \to \textbf{Real}$

  The result is the identity on the argument, just changing its type.

- **Cardinality of set:**

  $\textbf{card} : \mathrm{T}\textbf{-set} \stackrel{\sim}{\to} \textbf{Nat}$

  The result is the number of elements in the set.

- **Length of list:**

  $\textbf{len} : \mathrm{T}^{*} \to \textbf{Nat}$

  The result is the length of the list.

- **Indices of list:**

  $\textbf{inds} : \mathrm{T}^{\omega} \to \textbf{Nat-infset}$

  The result is the set of indices in the list. Let *f_list* be a finite list and let *i_list* be an infinite list, then:

  $\textbf{inds}\ \mathrm{f\_list} = \{\mathrm{n} \mid \mathrm{n} : \textbf{Nat} \bullet \mathrm{n} \geq 1 \wedge \mathrm{n} \leq \textbf{len}\ \mathrm{f\_list}\}$

  $\textbf{inds}\ \mathrm{i\_list} = \{\mathrm{n} \mid \mathrm{n} : \textbf{Nat} \bullet \mathrm{n} \geq 1\}$

- **Elements of list:**

  $\textbf{elems} : \mathrm{T}^{\omega} \to \mathrm{T}\textbf{-infset}$

  The result is the set of elements of the list.

- **Head of list:**

  $\textbf{hd} : \mathrm{T}^{\omega} \stackrel{\sim}{\to} \mathrm{T}$

Pre-condition: The list must be non-empty.

The result is the first element in the list.

- **Tail of list:**

  $\mathbf{tl} : \mathrm{T}^{\omega} \rightarrow \mathrm{T}^{\omega}$

  The result is the list which remains after removing the first element if present. The operator is the identity on the empty list.

- **Front of list:**

  $\mathbf{front} : \mathrm{T}^{*} \rightarrow \mathrm{T}^{*}$

  The result is the list which remains after removing the last element if present. The operator is the identity on the empty list.

- **Last of list:**

  $\mathbf{last} : \mathrm{T}^{*} \stackrel{\sim}{\rightarrow} \mathrm{T}$

  Pre-condition: The list must be non-empty.

  The result is the last element of the list.

- **Domain of map:**

  $\mathbf{dom} : (\mathrm{T}_1 \ \overrightarrow{_m}\ \mathrm{T}_2) \rightarrow \mathrm{T}_1\textbf{-infset}$

  The result is the domain of the map: the values for which it is defined.

- **Range of map:**

  $\mathbf{rng} : (\mathrm{T}_1 \ \overrightarrow{_m}\ \mathrm{T}_2) \rightarrow \mathrm{T}_2\textbf{-infset}$

  The result is the range of the map: the values that can be obtained by applying the map to the values in its domain.

12 Infix combinators

Syntax

infix_combinator ::=
  $\square$ |
  $\sqcap$ |
  $\|$ |
  $\#$ |
  ;

Meaning

The infix combinators are intended to compose expressions that either communicate or at least have side-effects on variables. Some simple proof-rules are associated with each combinator in order to clarify its semantics.

- **External Choice:**

  $\text{expr}_1 \;\square\; \text{expr}_2$

  An external choice is made between the effects of the two expressions. That is, the possible effect of a concurrently executing third expression can influence the choice.

  External choice has unit **stop**, has zero **chaos**, is idempotent, is commutative, is associative, and is distributive through internal choice:

  $\text{expr} \;\square\; \textbf{stop} \equiv$
    expr

  $\text{expr} \;\square\; \textbf{chaos} \equiv$
    $\textbf{chaos}$

  $\text{expr} \;\square\; \text{expr} \equiv$
    expr

  $\text{expr}_1 \;\square\; \text{expr}_2 \equiv$
    $\text{expr}_2 \;\square\; \text{expr}_1$

  $\text{expr}_1 \;\square\; (\text{expr}_2 \;\square\; \text{expr}_3) \equiv$
    $(\text{expr}_1 \;\square\; \text{expr}_2) \;\square\; \text{expr}_3$

  $\text{expr}_1 \;\square\; (\text{expr}_2 \sqcap \text{expr}_3) \equiv$
    $(\text{expr}_1 \;\square\; \text{expr}_2) \sqcap (\text{expr}_1 \;\square\; \text{expr}_3)$

- **Internal choice:**

  $\text{expr}_1 \sqcap \text{expr}_2$

  An internal – non-deterministic – choice is made between the effects of the two expressions. That is, the possible effect of a concurrently executing third expression cannot influence the choice.

  Internal choice has unit **swap**, has zero **chaos**, is idempotent, is commutative, and is associative:

  $\text{expr} \sqcap \textbf{swap} \equiv$
  $\quad \text{expr}$

  $\text{expr} \sqcap \textbf{chaos} \equiv$
  $\quad \textbf{chaos}$

  $\text{expr} \sqcap \text{expr} \equiv$
  $\quad \text{expr}$

  $\text{expr}_1 \sqcap \text{expr}_2 \equiv$
  $\quad \text{expr}_2 \sqcap \text{expr}_1$

  $\text{expr}_1 \sqcap (\text{expr}_2 \sqcap \text{expr}_3) \equiv$
  $\quad (\text{expr}_1 \sqcap \text{expr}_2) \sqcap \text{expr}_3$

- **Concurrent composition:**

  $\text{expr}_1 \parallel \text{expr}_2$

  The two expressions are made to execute concurrently with another. The two expressions can communicate through channels: one expression inputs from a channel which is output to by the other expression.

  Concurrent attempts to input from a channel and to output to the channel does, however, not necesarily lead to a communication. Whether it does, depends on an internal choice. The two expressions can thus communicate with a third expression which is concurrently composed with the two.

  Concurrent composition has unit **skip**, has zero **chaos**, is commutative, is associtive, and is distributive through internal choice:

  $\text{expr} \parallel \textbf{skip} \equiv$
  $\quad \text{expr}$

  $\text{expr} \parallel \textbf{chaos} \equiv$
  $\quad \textbf{chaos}$

expr$_1$ ∥ expr$_2$ ≡
  expr$_2$ ∥ expr$_1$

expr$_1$ ∥ (expr$_2$ ∥ expr$_3$) ≡
  (expr$_1$ ∥ expr$_2$) ∥ expr$_3$

expr$_1$ ∥ (expr$_2$ ⌈⌉ expr$_3$) ≡
  (expr$_1$ ∥ expr$_2$) ⌈⌉ (expr$_1$ ∥ expr$_3$)

The following two equivalences hold if the expression *s_expr* is a total one which also does not involve communication and if $c_1 \neq c_2$.

x:=c$_1$? ∥ c$_2$!s_expr ≡
  (x:=c$_1$? ; c2!s_expr) [] (c$_2$!s_expr ; x:=c$_1$?)

x:=c? ∥ c!s_expr ≡
  (((x:=c? ; c!s_expr) [] (c!s_expr ; x:=c?)) [] (x:=s_expr)) ⌈⌉ (x:=s_expr)

These are special cases of a more general law.

- **Interlocked composition:**

  expr$_1$ ╫ expr$_2$

  The two expressions are made to execute interlocked with another. The effect is similar to concurrent composition except that the two expressions are obliged to communicate exclusively with each other. Thus: concurrent attempts to input from a channel and to output to the channel does lead to a communication. On the other hand, if the two expressions only want to communicate, but not on the same channel, then the whole expression deadlocks.

  Interlocked composition has unit **skip**, has zero **chaos**, is commutative, and is distributive through internal choice:

  expr ╫ **skip** ≡
    expr

  expr ╫ **chaos** ≡
    chaos

  expr$_1$ ╫ expr$_2$ ≡
    expr$_2$ ╫ expr$_1$

  expr$_1$ ╫ (expr$_2$ ⌈⌉ expr$_3$) ≡
    (expr$_1$ ╫ expr$_2$) ⌈⌉ (expr$_1$ ╫ expr$_3$)

  The following two equivalences hold if the expression *s_expr* is a total one which does not involve communication, and if $c_1 \neq c_2$.

x:=c_1? ╫ c_2!expr $\equiv$
  stop

x:=c? ╫ c!s_expr $\equiv$
  x:=s_expr

As with the corresponding equivalences for concurrent composition, these are special cases of a more general law.

In general, the interlocking combinator illustrates the distinction between external choice and internal choice. The following equivalences hold if s_expr_1 and s_expr_2 are total expressions which do not involve communication, and if $c_1 \neq c_2$.

(x:=c_1? $\Box$ c_2!s_$expr_2$) ╫ c_1!s_$expr_1$ $\equiv$
  x := s_$expr_1$

(x:=c_1? $\sqcap$ c_2!s_$expr_2$) ╫ c_1!s_$expr_1$ $\equiv$
  (x := s_$expr_1$) $\sqcap$ **stop**

- **Sequential composition:**

$expr_1$; $expr_2$

The second expression is made to execute sequentially after the first expression. The value returned is the value returned by the second expression.

Sequential composition has unit **skip**, is associative, and is distributive on the right through internal choice:

expr ; **skip** $\equiv$
  expr

skip ; expr $\equiv$
  expr

$expr_1$; ($expr_2$; $expr_3$) $\equiv$
  ($expr_1$; $expr_2$) ; $expr_3$

($expr_1$ $\sqcap$ $expr_2$) ; $expr_3$ $\equiv$
  ($expr_1$; $expr_3$) $\sqcap$ ($expr_2$; $expr_3$)

13 Connectives

13.1 Infix connectives

Syntax

infix_connective ::=
  $\Rightarrow$ |
  $\vee$ |
  $\wedge$

Meaning

The infix connectives are intended to compose boolean expressions into new boolean expressions.

The effect of a composed expression follows a so-called conditional logic where in general the second constituent expression is evaluated only if the value of the first constituent expression is not enough to determine the value of the composed expression. In this way the eventual divergence, deadlock or default in the second constituent expression can be avoided when possible. The meaning of the connectives is given in terms of equivalences with the if expressions they are shorthands for.

- **And:**

  $\mathrm{expr}_1 \wedge \mathrm{expr}_2 \equiv$
    if expr_1 **then** expr_2 **else false end**

- **Or:**

  $\mathrm{expr}_1 \vee \mathrm{expr}_2 \equiv$
    if expr_1 **then true else** expr_2 **end**

- **Implies:**

  $\mathrm{expr}_1 \Rightarrow \mathrm{expr}_2 \equiv$
    if expr_1 **then** expr_2 **else true end**

13.2 Prefix connectives

Syntax

prefix_connective ::=

~ |
□

Meaning

The prefix connectives compose boolean expressions into new boolean expressions.

- **Not:**
  An axiom_prefix_expr of the form

  ~expr

  is a shorthand for

  if expr **then false else true end**

- **Always:**
  An axiom_prefix_expr of the form

  □ expr

  yields **true** if and only if for all states satisfying the subtype constraints for visible variables, the expr is convergent and deterministic and yields the value **true**.

  The axiom_prefix_expr itself is convergent and deterministic.

References

[1] *An RSL Tutorial*
RAISE/CRI/DOC/1/V1

[2] *RSL Proof Rules*
RAISE/CRI/DOC/5/V1

Index

A Lexical Matters

This section describes lexical matters, i.e. the micro-syntax for RSL.

Basically, RSL follows the rules now in current practise 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 zero 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 be nested.)

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

A.1 Varying Tokens

The micro-syntax for varying tokens is defined by the below syntax rules, 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

comment_char ::=
  LF |
  CR |
  TAB |

ascii_letter |
digit |
graphic |
prime |
quote

int_value_literal ::=
opt-sign digit-string

real_value_literal ::=
opt-sign digit-string '.' digit-string

text_value_literal ::=
'″' opt-text_character-string '″'

char_value_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 ::=
'α' | 'β' | 'γ' | 'δ' | 'ϵ' | 'ζ' | 'η' | 'θ' | 'ι' | 'κ' | 'μ' |
'ν' | 'ξ' | 'π' | 'ρ' | 'σ' | 'τ' | 'υ' | 'ϕ' | 'χ' | 'ψ' | 'ω' |
'Γ' | 'Δ' | 'Θ' | 'Λ' | 'Ξ' | 'Π' | 'Σ' | 'Υ' | 'Φ' | 'Ψ' | 'Ω'

underline ::=
‘_’

sign ::=
‘_’

prime ::=
‘'’

quote ::=
‘"’

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’

A.1.1 ASCII Forms of Greek Letters

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

ASCII	LaTeX	ASCII	LaTeX
‘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

A.2 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:

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

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

A.3 RSL keywords

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

Keywords for RSL			
Bool	**do**	**it**	**swap**
Char	**dom**	**last**	**then**
Int	**elems**	**len**	**tl**
Nat	**else**	**let**	**true**
Real	**elsif**	**local**	**type**
Text	**end**	**object**	**until**
Unit	**extend**	**of**	**use**
abs	**false**	**out**	**value**
any	**for**	**post**	**variable**
axiom	**forall**	**pre**	**while**
begin	**front**	**read**	**with**
card	**hd**	**rl**	**write**
case	**hide**	**rng**	**with**
channel	**if**	**scheme**	
chaos	**import**	**skip**	
class	**in**	**stop**	

B Precedence and associativity of operators

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

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

C Syntax summary

Specifications

specification ::=
  module_decl-string

module_decl ::=
  object_decl |
  scheme_decl

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]

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

Class expressions

class_expr ::=
  basic_class_expr |
  importing_class_expr |
  extending_class_expr |
  hiding_class_expr |

renaming_class_expr |
scheme_instantiation

Basic class expressions

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

Importing class expressions

importing_class_expr ::=
import object_expr-list **in** class_expr

Extending class expressions

extending_class_expr ::=
extend class_expr-list **with** opt-decl-string **end**

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)

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*-expr-list]

Array object expressions

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

Fitting object expressions

fitting_object_expr ::=
  object_expr renaming

Renamings

renaming ::=
  { rename_pair-list }

rename_pair ::=
  defined_item **for** defined_item

defined_item ::=
  id_or_op |
  disambiguated_item

disambiguated_item ::=
  id_or_op : type_expr

Declarations

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

Type declarations

type_decl ::=
type commented_type_def-list

commented_type_def ::=
opt-comment-string type_def

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

Sort definitions

sort_def ::=
id

Variant definitions

variant_def ::=
id == variant-choice

variant ::=
constant_variant |
record_variant

constant_variant ::=

constructor opt-subtype_naming

record_variant ::=
constructor component_kinds opt-subtype_naming

constructor ::=
id_or_op |
_

component_kinds ::=
(component_kind-list)

component_kind ::=
opt-destructor type_expr opt-reconstructor

destructor ::=
id_or_op :

reconstructor ::=
↔ id_or_op

subtype_naming ::=
@ id

Union definitions

union_def ::=
id = *type*-name-choice2

Short record definitions

short_record_def ::=
id :: component_kind-string

Abbreviation definitions

abbreviation_def ::=
id = type_expr

Value declarations

value_decl ::=
  value commented_value_def-list

commented_value_def ::=
  opt-comment-string value_def

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

Explicit value definitions

explicit_value_def ::=
  single_typing = *pure*-expr

Implicit value definitions

implicit_value_def ::=
  single_typing *pure*-restriction

Explicit function definitions

explicit_function_def ::=
  single_typing formal_function_application ≡ 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

pre_condition ::=
  pre *readonly_logical*-expr

Implicit function definitions

implicit_function_def ::=
  single_typing formal_function_application post_condition opt-pre_condition

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

result_naming ::=
  as binding

Variable declarations

variable_decl ::=
  variable commented_variable_def-list

commented_variable_def ::=
  opt-comment-string variable_def

variable_def ::=
  single_variable_def |
  multiple_variable_def

single_variable_def ::=
  id : type_expr opt-initialisation

initialisation ::=
  := *pure*-expr

multiple_variable_def ::=
  id-list2 : type_expr

Channel declarations

channel_decl ::=
  channel commented_channel_def-list

commented_channel_def ::=
  opt-comment-string channel_def

channel_def ::=
  single_channel_def |
  multiple_channel_def

single_channel_def ::=
  id : type_expr

multiple_channel_def ::=
  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 *pure_logical*-expr

axiom_naming ::=
  [id]

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 |
bracketted_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$^\omega$

Map type expressions

map_type_expr ::=
  type_expr $\underset{m}{\rightarrow}$ type_expr

Function type expressions

function_type_expr ::=
  type_expr function_arrow result_desc

function_arrow ::=
  $\stackrel{\sim}{\rightarrow}$ |
  $\rightarrow$

result_desc ::=
  opt-access_desc-string type_expr

Access descriptions

access_desc ::=
  access_mode access-list

access_mode ::=
  read |
  write |
  in |
  out

access ::=

variable_or_channel-name |
completed_access |
comprehended_access

completed_access ::=
opt-qualification **any**

comprehended_access ::=
{ access-list | *pure*-set_limitation }

Subtype expressions

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

Bracketted type expressions

bracketted_type_expr ::=
(type_expr)

Expressions

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 |
  bracketted_expr |
  infix_expr |
  prefix_expr |
  comprehended_expr |
  initialise_expr |
  assignment_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 ::=
  (expr-list2)

Set expressions

set_expr ::=
  ranged_set_expr |
  enumerated_set_expr |
  comprehended_set_expr

Ranged set expressions

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

Enumerated set expressions

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

Comprehended set expressions

comprehended_set_expr ::=
  { *readonly*-expr | set_limitation }

set_limitation ::=
  typing-list opt-restriction

restriction ::=
  • *readonly_logical*-expr

List expressions

list_expr ::=
  ranged_list_expr |
  enumerated_list_expr |
  comprehended_list_expr

Ranged list expressions

ranged_list_expr ::=
  ⟨ *readonly_integer*-expr .. *readonly_integer*-expr ⟩

Enumerated list expressions

enumerated_list_expr ::=
  ⟨ *readonly*-opt-expr-list ⟩

Comprehended list expressions

comprehended_list_expr ::=
  ⟨ *readonly*-expr | list_limitation ⟩

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

Map expressions

map_expr ::=
  enumerated_map_expr |
  comprehended_map_expr

Enumerated map expressions

enumerated_map_expr ::=
  [opt-expr_pair-list]

expr_pair ::=
  readonly-expr $\mapsto$ *readonly*-expr

Comprehended map expressions

comprehended_map_expr ::=
  [expr_pair | set_limitation]

Function expressions

function_expr ::=
  λ lambda_parameter • expr

lambda_parameter ::=
  lambda_typing |
  single_typing

lambda_typing ::=
  (opt-typing-list)

Application expressions

application_expr ::=
  list_or_map_or_function-expr actual_function_parameter-string

actual_function_parameter ::=
  (opt-expr-list)

Quantified expressions

quantified_expr ::=
  quantifier typing-list restriction

quantifier ::=
  ∀ |
  ∃ |
  ∃!

Equivalence expressions

equivalence_expr ::=
  expr ≡ expr opt-pre_condition

Post expressions

post_expr ::=
  expr post_condition opt-pre_condition

Disambiguation expressions

disambiguation_expr ::=
  expr : type_expr

Bracketted expressions

bracketted_expr ::=
  (expr)

Infix expressions

infix_expr ::=
  stmt_infix_expr |
  axiom_infix_expr |
  value_infix_expr

Stmt infix expressions

stmt_infix_expr ::=
  expr infix_combinator expr

Axiom infix expressions

axiom_infix_expr ::=
  logical-expr infix_connective *logical*-expr

Value infix expressions

value_infix_expr ::=
  expr infix_op expr

Prefix expressions

prefix_expr ::=
  axiom_prefix_expr |
  value_prefix_expr

Axiom prefix expressions

axiom_prefix_expr ::=
  prefix_connective *logical*-expr

Value prefix expressions

value_prefix_expr ::=
  prefix_op expr

Comprehended expressions

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

Initialise expressions

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

Assignment expressions

assignment_expr ::=
  variable-name := expr

Input expressions

input_expr ::=
  channel-name ?

Output expressions

output_expr ::=
  channel-name ! expr

Structured expressions

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

Local expressions

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

Let expressions

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

let_def ::=
  typing |
  explicit_let |
  implicit_let

explicit_let ::=
  let_binding = expr

implicit_let ::=
  single_typing restriction

let_binding ::=
  binding |
  record_pattern |
  list_pattern

If expressions

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

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

else_branch ::=
  else expr

Case expressions

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

case_branch ::=
  pattern → expr

For expressions

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

While expressions

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

Until expressions

until_expr ::=
  do *unit*-expr **until** *logical*-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

Patterns

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

Wildcard patterns

wildcard_pattern ::=
  _

Product patterns

product_pattern ::=
  (pattern-list2)

Record patterns

record_pattern ::=
  pure_value-name component_patterns

component_patterns ::=
  (inner_pattern-list)

inner_pattern ::=
  binding |
  wildcard_pattern

List patterns

list_pattern ::=
  constructed_list_pattern |
  left_list_pattern |
  right_list_pattern |
  left_right_list_pattern

Constructed list patterns

constructed_list_pattern ::=
  ⟨ opt-inner_pattern-list ⟩

Left list patterns

left_list_pattern ::=
  constructed_list_pattern ⌢ id_or_wildcard

id_or_wildcard ::=
  id |
  wildcard_pattern

Right list patterns

right_list_pattern ::=
  id_or_wildcard ⌢ constructed_list_pattern

Left right list patterns

left_right_list_pattern ::=
  constructed_list_pattern ⌢ id_or_wildcard ⌢ constructed_list_pattern

Names

name ::=
  qualified_id |
  qualified_op

Qualified ids

qualified_id ::=
  opt-qualification id

qualification ::=
  element-object_expr .

Qualified ops

qualified_op ::=
  opt-qualification (op)

Identifiers and operators

id_or_op ::=
  id |
  op

op ::=
  infix_op |
  prefix_op

Infix ops

infix_op ::=
  $=$ |
  $\neq$ |
  $>$ |
  $<$ |
  $\geq$ |
  $\leq$ |
  $\supset$ |

$\subset$ |
$\supseteq$ |
$\subseteq$ |
$\in$ |
$\notin$ |
$+$ |
$-$ |
$\backslash$ |
$\widehat{\ }$ |
$\cup$ |
$\dagger$ |
$*$ |
$/$ |
$\circ$ |
$\cap$ |
$\uparrow$ |
$\$$

Prefix ops

prefix_op ::=
abs |
it |
rl |
card |
len |
inds |
elems |
hd |
tl |
front |
last |
dom |
rng

Connectives

connective ::=
  infix_connective |
  prefix_connective

Infix connectives

infix_connective ::=
  $\Rightarrow$ |
  $\vee$ |
  $\wedge$

Prefix connectives

prefix_connective ::=
  $\sim$ |
  $\square$

Infix combinators

infix_combinator ::=

  $[\!]$ |

  $\lceil\!\rceil$ |

  $\|$ |

  $+\!\!+$ |

  ;