SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 99-21

The Sum Reference Manual $v1.4$

Wendy Johnston and Luke Wildman

November, 1999

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

The Sum Reference Manual $v1.4$

Wendy Johnston and Luke Wildman

Document Control

Document History

v1.4 (February 1999) This version coincides with the release Cogito1.2 tools. The Sum language of Cogito1.2 has been extended to include an intermediate language to facilitate translation of a Sum specification to executable code.
Editors: Wendy Johnston, Luke Wildman.

v1.3 (15th May 1997) This version, a major revision, to coincide with release of Cogito tools for Windows. New sections on module contents, predicates and expressions.
Principal Editors: Peter Kearney, Andrew Martin, Kelvin Ross.

v1.2 (16th April 1996) Coincided with 1996 Sum training courses. Added sections on logic and arithmetic.
Principal Editor: Nicholas Hamilton.

v1.0 (December 1994) Produced as part of preparation for the first Sum training courses.
Principal Editors: Anthony Bloesch, Ed Kazmierczak, Peter Kearney, Owen Traynor, et al.

Changes Forecast Will continue to be updated in line with enhancements to the Sum language and the Cogito toolset.

Recommendations for change to: `cogito@svrc.uq.edu.au`.

Contents

List of Figures

1. Introduction

Cogito is a methodology and associated toolset for the formal development of software. The Cogito specification language (Sum) is based on a simplified version of the Z [19] specification language, enhanced with constructs that support formal software development. The most notable of these enhancements are a module mechanism and the explication of the intended purpose of a schema (i.e. operation, state, and initialisation).

Fully formal development in Cogito is targeted to an intermediate programming language (IL), defined as a subset of Sum, which can be translated to a variety of imperative languages. For a further discussion on this see B.4.

Chapter Organisation Chapter 2 describes the definitional facilities of Sum: modules, schemas, axiomatic definitions, given sets, etc. Chapter 3 describes the predicates which may be written in schemas and axioms. Chapter 4 gives details of the general forms of expressions which may appear in those predicates. Chapter 5 defines a collection of mathematical functions, relations and datatypes (the 'mathematical toolkit') which can be used to construct specifications.

Appendices The first appendix summarises the syntax currently supported by the Sum toolset, and appendix C reproduces the Sum definition file for the mathematical toolkit. Appendix D tabulates the correspondences between the ASCII and mathematical forms of the language (see below). An account of the Cogito methodology, with particular emphasis on the role of modules is in Appendix B. Appendix E outlines the chief differences between Sum and Z.

Note on concrete syntax. Sum has two forms. The 'mathematical' form resembles Z and traditional mathematics and is useful for hand-written work. The 'ASCII' form is more amenable to typing on a computer. In most places, this manual presents both forms. In some cases where the two are identical, and in certain examples, one or other form is omitted.

The symbols of the ASCII form are presented in `typewriter` font.

Appendix A presents the full concrete syntax for the ASCII form. All input to the Cogito toolset takes place using this form. The mathematical form is used here for presentation purposes, and is available as output from certain of the Cogito tools. A table of correspondences between ASCII and mathematical forms is given in Appendix D.

The mathematical toolkit of Chapter 5 is largely presented using Sum. The definitions are written using the mathematical form, and occasionally make use of Z-like facilities which are not available when writing Sum specifications. For example, certain infix generics are defined using the abbreviation notation. Many of these definitions are accompanied by mathematical laws which document their interrelationships. Some of these are indicative only; formal details are suppressed by use of ellipses.

2. Modules and their contents

This chapter describes the means of making declarations in Sum, and how to group them together into modules. Appendix B offers a more discursive account of this material, and describes how it is used in the Cogito methodology.

2.1 Module Definitions

2.1.1 Simple Module

Syntax

Id
DeclarationList

`module` *Id* `is` *DeclarationList* `end` *Id*

Description This is the simplest special case of the more general module declarations which follow. The module name must be unique. The contents of the module are a sequence of declarations (See Section 2.3), separated by semicolons.

Example This module declares two functions on integers, which respectively return the square and the square root of their arguments.

Squaring

$$sqr, sqrt : \mathbb{N} \to \mathbb{N}$$

$$\forall x : \mathbb{N} \bullet sqr(x) = x * x$$
$$\forall x, k : \mathbb{N} \bullet sqr(k) \leq x \land sqr(k+1) > x \Rightarrow sqrt(x) = k$$

```
module Squaring is
  axiom is
  dec
    sqr, sqrt: nat --> nat
  pred
    forall x: nat @ sqr(x) = x*x;
    forall x, k: nat @
      sqr(k) <= x and sqr(k+1) > x
      => sqrt(x) = k
  end
end Squaring
```

2.1.2 Parametrised Module

Syntax

Id[*ParamList*]
DeclarationList

`module` *Id*(*ParamList*) `is` *DeclarationList* `end` *Id*

Description In addition to the description for simple modules (page 3), this form defines a module which can later be instantiated by supplying parameters. The formal parameters in the *ParamList* are separated by semicolons. They may be either single identifiers (in which case they denote *sets* which the module may use like given sets (see Section 2.3.1) or variable declarations of the form

DeclName, . . . , *DeclName* : *Expression*

(in which case they supply individual values for use in the module). Notice that in the second case, the *Expression* must belong to one of Sum's built-in types (integers, strings, etc.), be a parameter appearing previously in the list, or an expression constructed from these using the toolkit operators.

Example This example declares a module which defines a function *sorted*. This function determines whether or not a given sequence is sorted. The module parameters are *Item* (the set (type) from which the sequence elements are drawn) and *ord* (an ordering relation on elements of *Item*).

Sort[*Item*; *ord* : *Item* × *Item* → $\mathbb{B}$]

$$sorted : \operatorname{seq} Item \to \mathbb{B}$$

$$sorted(\langle\rangle)$$
$$\forall L : \operatorname{seq} Item \bullet L \neq \langle\rangle \Rightarrow sorted(L) \Leftrightarrow (\forall i, j : \operatorname{dom} L \bullet i < j \Rightarrow ord(L(i), L(j)))$$

```
module Sort
 (Item;
  ord: Item cross Item --> bool)
is
  axiom is
  dec
    sorted: seq Item --> bool
  pred
    sorted(<>);
    forall L: seq Item @
      L /= <> => sorted(L) <=>
      (forall i, j: dom L @
         i < j => ord (L(i), L(j)) )
  end
end Sort
```

2.1.3 Parametrised Module with Parameter Constraints

Syntax

Id[*ParamList* | *PredList*]
DeclarationList

`module` *Id*(*ParamList*) [*PredList*] `is` *DeclarationList* `end` *Id*

Description This form is like that for parametrised modules (page 4), with an additional sequence of semicolon-separated predicates. These predicates constrain the permissible values of the parameters.

Example This example declares a module which defines a function *sorted*. This function determines whether or not a given sequence is sorted. The module parameters are *Item* (the set (type) from which the sequence elements are drawn) and *ord* (an ordering relation on elements of *Item*).

The ordering relation *ord* is constrained; it must be *antireflexive*.

$$SortAR\ [Item;\ ord : Item \times Item \to \mathbb{B} \mid \forall i, j : Item \bullet ord(i,j) \Rightarrow \neg\ (ord(j,i))]$$

$$sorted : \mathrm{seq}\ Item \to \mathbb{B}$$

$$sorted(\langle\,\rangle)$$
$$\forall L : \mathrm{seq}\ Item \bullet L \neq \langle\,\rangle \Rightarrow sorted(L) \Leftrightarrow (\forall i, j : \mathrm{dom}\ L \bullet i < j \Rightarrow ord(L(i), L(j)))$$

```
module SortAR
 (Item;
  ord: Item cross Item --> bool)
 [forall i,j:Item @ ord(i,j) =>
                    not(ord(j,i))]
is
  axiom is
  dec
    sorted: seq Item --> bool
  pred
    sorted(<>);
    forall L: seq Item @
      L /= <> => sorted(L) <=>
      (forall i, j: dom L @
         i < j => ord (L(i), L(j)) )
  end
end SortAR
```

2.2 Module Imports

2.2.1 Import

Syntax

import Id `import` *Id* – simple import

Description *import* is a declaration paragraph which may appear anywhere within a Sum module. It allows declarations from one module to be *shared* in another. The declaration

import M

behaves exactly as if the contents of module *M* were inserted at this position in the specification, *except* that care must be taken with the names of identifiers.

Any names (schemas, schema components, functions, etc.) declared in module *M* gain an *M* prefix when used in another module. Thus, if module *M* declares schema *S*, and module *N* contains *import M*, then the schema declared as *S* must be referred to in module *N* as *M.S*.

Module import is transitive. If module *M* is imported into module *N*, and module *N* is imported into module *P*, then the defninitions from *M* are also imported into *P*. In this case, a schema *S* declared in *M* must be referred to in module *P* as *N.M.S*.

Only simple (non-parametrised) modules may be used in this simple import (see *import as*), page 7).

Example The following module imports *Squaring* (see page 3), and then uses the *sqr* function (as *Squaring.sqr*) to define a function for computing an area.

Circle

import Squaring

$$area : \mathbb{N} \to \mathbb{N}$$

$$\forall radius : \mathbb{N} \bullet area(radius) = Squaring.sqr(radius) * 22 \operatorname{div} 7$$

```
module Circle is
  import Squaring;
  axiom is
  dec
    area: nat --> nat
  pred
    forall radius: nat @ area(radius) =
      Squaring.sqr(radius) * 22  div 7
  end
end Circle
```

2.2.2 Import-as

Syntax

import Id as Id	– module copying
import Id(*ExpressionList*) *as Id*	– parameter instantiation
import Id{*RenameList*} *as Id*	– component renaming
import Id(*ExpressionList*) {*RenameList*} *as Id*	– all of the above

Description Import-as creates a *copy* of the specified module, by using the supplied name to qualify the names (compare the description of *import*, page 6), not the original module name. If the module is parametrised, actual parameters must be supplied. Names declared in the imported module may be renamed.

As with *import* (page 6), *import as* is transitive. Notice that when a module *M* is imported into modules *N* and *P*, if *N* and *P* are 'imported *as*' in some other module, then they are *copied* whereas the definitions from *M* are *shared*.

Examples The following module imports the *Sort* module (see page 4), instantiating it for the natural numbers and the *leq* relation, and renaming the *sorted* function as *natsorted*.

UseSort

import *Sort*($\mathbb{N}$, $\leq$){*natsorted/sorted*} *as SortNats*;

⋮

```
module UseSort is
  import Sort(nat, (_<=_))
    {natsorted/sorted} as SortNats
  ⋮
end UseSort
```

2.2.3 Visible

Syntax

visible Id	– make module visible
visible Id{*Selections*}	– make selected identifiers visible

Description The visible declaration makes entities in an imported module directly accessible without the need for a qualified name. It does not make visible the entities of transitively imported modules. Thus in Sum the visible declaration is *non-transitive*.

In the second form, only those identifiers from the specified module which are listed in the *Selections* set are made visible.

Visibility of imported entities is *linear*. Given two entities which are in scope and which have the same unqualified name the directly visible entity is the most recently declared.

Examples In module *Cylinder* the addition of the declaration *visible Circle* means that we can reference the entity *area* in *Circle* directly without qualification but we cannot access the entities *sqr* and *sqrt* of *Squaring* directly. *Squaring* is transitively imported into *Cylinder* but is not transitively visible. The entities *sqr* and *sqrt* however, are in scope and may be accessed via qualified names.

Definitions of *Squaring* and *Circle* are found on pages 3 and 6 respectively.

Cylinder

import Circle
visible Circle

$$length : \mathbb{N}$$

$$area : \mathbb{N} \to \mathbb{N}$$

$$\forall radius : \mathbb{N} \bullet area(radius) = 2 * area(radius) + (2 * radius * length * 22) \operatorname{div} 7$$

```
module Cylinder is

  import Circle;
  visible Circle;

  axiom is
  dec
    length: nat
  end;

  axiom is
  dec
    area: nat --> nat
  pred
    forall radius: nat @
      area(radius) = 2*area(radius)+
          (2*radius*length*22) div 7
  end

end Cylinder
```

2.3 Declarations

2.3.1 Given Sets

Syntax

[*DeclNameList*]

Description This declaration introduces one or more given sets, that is, sets for which no internal structure is specified.

Example

$[AcctNos, Customers]$	`[AcctNos,Customers]`

2.3.2 Abbreviations

Syntax

Id == *Expression*
Id == (*SchemaExpression*)

Description An abbreviation declaration introduces a new identifier and defines its value.
Schema expressions must be bracketed. Note that generic schema cannot be defined this way.

Examples

$ST == SYM \pfun VAL$	`ST == SYM -||-> VAL;`
$max == 100$	`max == 100;`
$tot\_elems == rows * cols$	`tot_elems == rows*cols;`
$S == ([x : nat \mid x > 1])$	`S == ([x:nat|x>1]);`
$S == (U \lor V)$	`S == (U or V);`

2.3.3 Axiom Declaration

Syntax

VarDeclList
PredicateList

`axiom is` [`dec` *VarDeclList*] [`pred` *PredicateList*] `end`

Description An axiom introduces one or more constants with their types, and asserts a predicate which may constrain the constants introduced.

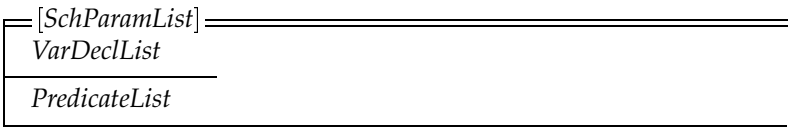
Examples

$small, large : \mathbb{N}$
$small < large$

```
axiom is
dec
   small, large: nat
pred
   small < large
end
```

2.3.4 Generic Axiom Declaration

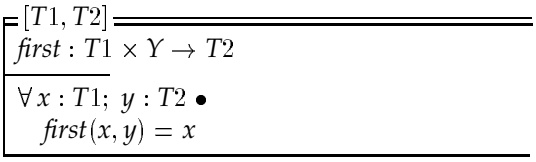
Syntax




```
axiom [SchParamList] is
        [dec VarDeclList] [pred PredicateList] end
```

Description A generic axiom is parametrised with one or more generic types. This enables the declarations and properties to be given uniformly for a number of type parameters.

Examples




```
axiom [T1, T2] is
dec
    first: T1 cross T2 --> T1
pred
   forall x: T1;y:T2 @
                first(x,y) = y
end
```

2.3.5 Variable Declaration

Syntax

DeclNameList : *Expression*

Description This declaration introduces a constant and its type. It has the same effect as an axiom without the predicate part.

Examples

$maxsize : \mathbb{N}$	`maxsize: nat`

2.3.6 Predicate Constraints

Syntax

Predicate

See Chapter 3 for a description of predicates.

Description A predicate may appear as a declaration. Its effect is to specify a constraint on a previously declared constant.

Example

```
maxsize <= k
```

2.3.7 Free type definitions

Syntax

$$Id ::= Id \ [\langle\langle Expression \rangle\rangle] \ \{ \mid Id \ [\langle\langle Expression \rangle\rangle]\}$$

Description Free types allow a new set to be introduced as well as defining constructors to generate elements of the type. The simplest free types simply declare a set and its members. E.g.

$$colours ::= green \mid orange \mid red$$

This is equivalent to a definition

$$[colours]$$

$$\begin{array}{|l} green, orange, red : colours \\ \hline \forall c : colours \bullet \\ \quad c = green \lor c = orange \lor c = red \end{array}$$

A constructor may be either an identifier, denoting an element of the new type (as above), or a constructor function which is a function taking an argument and returning an element of the new type. Distinct constructors denote distinct elements and distinct values of arguments to constructor functions return distinct elements of the free type. The constructors generate all elements of the type.

Examples We might want a single type that can hold *either* a name or a number:

$$name\_or\_num ::= name\langle\langle NAME\rangle\rangle \mid number\langle\langle \mathbb{N} \rangle\rangle$$

Free types can be used to define recursive types such as trees.

E.g.

$$TREE ::= tip \mid fork\langle\langle \mathbb{N} \times TREE \times TREE\rangle\rangle$$

This declares a set *TREE*, with an element *tip*, and a function $fork : \mathbb{N} \times TREE \times TREE \rightarrow TREE$. It also says that every member of *TREE* is either a *tip* or has the form $fork(n, t_1, t_2)$ (but not both).

2.3.8 Enumerated type definition

Syntax

Id `::=` `enum` (*Id* { `,` *Id* })

Description

E `::=` `enum` (*C*1, . .*Cn*)

declares the type *E* consisting of the constants *C*1, . .*Cn*.

An enumerated type is equivalent to the following definitions in Sum:

E ::= *C*1 | *C*2 | . . | *Cn*;
`import` `adaenum` (*E*) `as` *E_enum*;
`visible` *E_enum*;

where the module `adaenum` includes declarations for each of the required ordering operators. See B.4.3 for more details.

Example With the definition,

Grades ::= `enum` (*fail*, *pass*, *credit*, *distinction*);

the following predicates are true:

fail < *pass*
pass ≤ *distinction*
credit > *pass*
credit ≥ *fail*
distinction = *distinction*
fail ≠ *pass*
succpass = *credit*

2.3.9 Schema definitions

Syntax

[op]Id[SchParamList]
BasicDeclList

PredicateList

[op] `schema` *Id* [`[` *SchParamList* `]`] `is` [`dec` *BasicDeclList*]
[`pred` *PredicateList*] `end` *Id*

$S == (SchemaExpression)$

Description A schema consists of a *signature*, which declares one or more variables and a predicate over those variables. The schema is named, and so may be referred to elsewhere in a specification. A schema may also have one or more generic type parameters.

The optional keyword `op` signifies (if used) that the schema is an *operation* schema. This results in both dashed and undashed copies of the *state* schema being automatically included in the schema. The use of **op** also signifies that this is an operation of the associated state machine. This has significance for the generation of validation and data refinement proof obligations. See appendix B for a fuller discussion of state and operation schemas.

The in-line alternative definition applies only to non-generic schema and is described in 2.3.2.

Examples *Library* is an example of a simple schema definition.

library
$held, on\_shelves : \mathbb{P}\, Books$
$borrowers : \mathbb{P}\, People$
$on\_loan : Book \pfun People$

$\mathrm{dom}\, on\_loan \subseteq held$
$\mathrm{ran}\, on\_loan \subseteq borrowers$
$on\_shelves \cup \mathrm{dom}\, on\_loan = held$

```
schema library is
dec
   held, on_shelves: power Books;
   borrowers: power People;
   on_loan: Book -|-> People
pred
   dom on_loan subset held;
   ran on_loan subset borrowers;
   on_shelves union dom on_loan
                          = held
end library
```

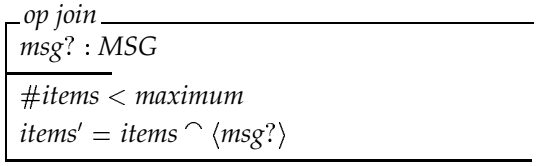
If we have the *state* schema

state
$items : \mathrm{seq}\, MSG$

$\#items \leq maximum$

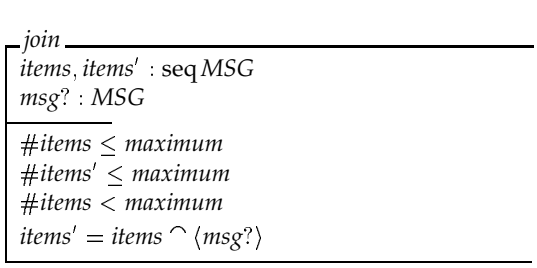
```
schema state is
dec
   items: seq MSG
pred
   #items <= maximum
end buffer_state;
```

then the operation schema




```
op schema join is
dec
   msg?:MSG
pred
   #items < maximum;
   items' = items^<msg?>
end join
```

is equivalent to




```
schema join is
dec
   items, items': seq MSG;
   msg?: MSG
pred
   #items <= maximum;
   #items' <= maximum;
   #items < maximum;
   items' = items ^ <msg?>
end join;
```

2.3.10 Function definition for the intermediate language

Syntax

`function` [*VarDeclList*] [`pred` *PredList*] `end`

Description The function definition is part of the intermediate language (described in B.4. A function in the intermediate language corresponds to an axiom with certain restrictions. These restrictions are as follows:

1. The *VarDeclList* must be the declaration of a function whose domain is expressed as either a set or a Cartesian product, and whose range is a set.
2. The *PredList* must be a quantified predicate. The characteristic tuple type of the quantifiable variables must be the same as that of the domain of the function declared in the *VarDeclList*.
3. Within the predicate of the quantified predicate, there must be a predicate of the form

   $$function\_application = expression$$

   where the type of *expression* is that of the range of the domain of the function declared in the *VarDeclList*, and *function_application* is an application of the name of the function declared in the *VarDeclList* to the characteristic tuple of the quantified variables.

Example

```
function f : ( integer cross integer ) -|-> integer
pred
    forall a : integer ; b : integer @
        f(a, b) = a + b
end
```

This definition is translated to a function in Ada:

```
function f (a : integer ; b : integer ) is integer
begin
    return a + b
end
```

2.3.11 Ergo declarations

Syntax

(`-+ ergo axiom` *Id* `+-`)
(`--:refinement:` *commands* `--`)
(`--:safety:` *commands* `--`)

Description When translating a Sum specification to a theory for use with the Ergo theorem prover, the axioms of the theory generated from Sum axioms are named using the prefix *axiom* and appending a (unique) number. For the purpose of improving readability, the declaration, (`-+ ergo axiom` *Id* `+-`), changes the default prefix to *Id*.

Sum users can pass on explicit Ergo commands at the end of the Ergo translation using the (`--: refinement:` *commands* `--`) declaration for refinement commands, or the (`--:safety:` *commands* `--`) declaration for other safety commands.

Example

(`-+ ergo axiom` *Key* `+-`)
(`--:refinement:` *data_refine*('*old_key*','*new_key*', ['*DoorOpen*'], *protocolReln*). `--`)
(`--:safety:` *macro*('*Key.thy*'). `--`)

State machine

Syntax

`state_machine` == `module` (*Id* , *Id*, [*DeclNameList*])

Description This is a declaration to the theorem prover, Ergo, that the state machine is the module whose declarations are the names in the parameter list. The names in the parameter list are, in order, the name of the state schema, the name of the initialising schema and the names of the operation schemas.

By default, the `state_machine` includes the explicitly declared state and init operations and all explicitly declared operations in the specification. This definition overrides that default.

It is used by ergo to generate proof obligations and by the Ada translator to identify operations for translation.

Example

`state_machine` == `module`
(*BirthdayBook.state* , *BirthdayBook.init* , *BirthdayBook.AddBirthday* , *BirthdayBook.FindBirthday*)

3. Predicates

3.1 Logic

Sum is based on first-order classical logic. See [3, 14] for a good introduction to logic.

3.1.1 Propositional Logic

Syntax

$\mathbb{B}$	`bool`	–Set of Boolean Values
Predicate $\wedge$ *Predicate*	*Predicate* `and` *Predicate*	–Logical conjunction
Predicate $\vee$ *Predicate*	*Predicate* `or` *Predicate*	–Logical disjunction
$\neg$ *Predicate*	`not` *Predicate*	–Logical negation
Predicate $\Rightarrow$ *Predicate*	*Predicate* `=>` *Predicate*	–Logical implication
Predicate $\Leftrightarrow$ *Predicate*	*Predicate* `<=>` *Predicate*	–Logical equivalence
true	`true`	–truth
false	`false`	–falsehood

Description
$\mathbb{B}$ is the set of boolean values, i.e. the set containing 'true' and 'false'. The operators are the usual logical operators of propositional logic.

Laws

$\text{true} : \mathbb{B}$
$\text{false} : \mathbb{B}$
$(A \wedge B) : \mathbb{B}$
$(A \vee B) : \mathbb{B}$
$(\neg A) : \mathbb{B}$
$(A \Rightarrow B) : \mathbb{B}$
$(A \Leftrightarrow B) : \mathbb{B}$
$(\text{true} \wedge \text{true}) = \text{true}$
$(\text{true} \wedge \text{false}) = \text{false}$
$(\text{false} \wedge \text{true}) = \text{false}$
$(\text{false} \wedge \text{false}) = \text{false}$
$(\text{true} \vee \text{true}) = \text{true}$
$(\text{true} \vee \text{false}) = \text{true}$
$(\text{false} \vee \text{true}) = \text{true}$
$(\text{false} \vee \text{false}) = \text{false}$
$(\neg\ \text{true}) = \text{false}$
$(\neg\ \text{false}) = \text{true}$
$(\text{true} \Rightarrow \text{true}) = \text{true}$
$(\text{true} \Rightarrow \text{false}) = \text{false}$
$(\text{false} \Rightarrow \text{true}) = \text{true}$
$(\text{false} \Rightarrow \text{false}) = \text{true}$
$A \vee \neg A$
$\neg (\neg P) \Leftrightarrow P$
$P \Rightarrow Q \Leftrightarrow \neg (P \wedge \neg Q)$
$\neg (A \wedge B) \Leftrightarrow \neg A \vee \neg B$
$(A \Rightarrow B) \vee (B \Rightarrow A)$
$(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$
$(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$
$(A \Leftrightarrow B) \Leftrightarrow (A \vee \neg B) \wedge (B \vee \neg A)$
$A \vee (\neg A \vee B) \Leftrightarrow \text{true}$
$A \wedge (\neg A \wedge B) \Leftrightarrow \text{false}$
$A \wedge (B \wedge C) \Leftrightarrow A \wedge (C \wedge B)$
$A \vee (B \vee C) \Leftrightarrow A \vee (C \vee B)$
$A \wedge B \Rightarrow B$

3.1.2 Predicate Logic

Syntax

$\forall$ *VarDeclList* $\bullet$ *Predicate*	`forall` *VarDeclList*`@`*Predicate*	– Universal quantification
$\exists$ *VarDeclList* $\bullet$ *Predicate*	`exists` *VarDeclList*`@`*Predicate*	– Existential quantification
$\exists_1$ *VarDeclList* $\bullet$ *Predicate*	`exists_1` *VarDeclList*`@`*Predicate*	– Unique quantification

Description
These are quantifiers of typed predicate logic.

Laws

$\forall x : T \bullet \text{true} \Leftrightarrow \text{true}$
$\neg\ (\forall x : T \bullet A) \Leftrightarrow (\exists x : T \bullet (\neg\ A))$
$\neg\ (\forall x : T \bullet A) \Leftrightarrow (\exists x : T \bullet (\neg\ A))$
$\forall x : T \bullet (A \Rightarrow B) \Rightarrow (\forall x : T \bullet A \Rightarrow \forall x : T \bullet B)$
$\forall x : T \bullet (A \wedge B) \Leftrightarrow (\forall x : T \bullet A) \wedge (\forall x : T \bullet B)$
$\exists x : T \bullet (A \wedge B) \Rightarrow (\exists x : T \bullet A) \wedge (\exists x : T \bullet B)$
$\exists x : T \bullet (A \vee B) \Leftrightarrow (\exists x : T \bullet A) \vee (\exists x : T \bullet B)$
$X : T \wedge [X/x]A \Rightarrow \exists x : T \bullet A$
$\forall x : T \bullet A \Rightarrow (X : T \Rightarrow [X/x]A)$
$\forall x : T \bullet A \Rightarrow \exists x : T \bullet A$
$\exists x : T \forall y : S \bullet A \Rightarrow \forall y : S \exists x : T \bullet A$

3.1.3 Definedness

Syntax

Expression↓ `def` *Expression* – assert that something is defined

Description Undefined terms can result from improper mu terms (see section 4.9, page 47) and from the application of a partial function outside its domain (section 4.11, page 49). For an expression *e*, the predicate *e*↓ is an assertion that *e* is defined.

Laws

$\mathbb{B}\downarrow$
$\mathbb{N}\downarrow$
true$\downarrow$
false$\downarrow$

Note It is generally poor style to write a specification which depends on definedness checking.

3.2 Specification Constructs

3.2.1 Explicit Precondition

Syntax

pre *Predicate* `pre` *Predicate*

Description This construct is used to indicate the explicit precondition of an operation schema (see Section 2.3.9). Its argument is a predicate.

Note $\mathrm{pre}(P)$ is logically equivalent to P. The annotation 'pre' has an effect *only* in determining the proof obligations for a specification.

An operation schema may contain at most one 'pre' statement.

When one schema is included in another (section 4.17.6, page 60), the 'pre' annotation is not included (though the predicate which it labels *is*).

Example This schema declares an update operation and claims that a sufficient precondition is $\#\,\mathrm{dom}(st) < k \land s? \notin \mathrm{dom}(st)$.

op update

$s? : SYM$

$v? : VAL$

$st, st' : SType$

$\mathrm{pre}(\#\,\mathrm{dom}(st) < k \land s? \notin \mathrm{dom}(st))$

$st' = st \oplus \{s? \mapsto v?\}$

$\#\,\mathrm{dom}(st) \leq k$

$\#\,\mathrm{dom}(st') \leq k$

```
op schema update is
dec
  s? : SYM;
  v? : VAL
pred
  pre(#dom(st) < k
      and s? not_in dom(st) );
  st' = st
        func_override { s? |--> v? };
  changes_only {st}
end update
```

(The module definition from which this schema is taken is printed in full on page 124.)

3.2.2 Variables Changed by an Operation

Syntax

$changes\_only\{x_1, \ldots x_n\}$ `changes_only{x1,...,xn}`

Description This predicate is used in the context of an operation schema to indicate which of the variables of the schema are changed by the operation. $changes\_only(S)$ asserts that for all variables v in the signature (see section 2.3.9) of the schema (which for an operation schema implicitly includes both *state* and *state'*) such that there is a dashed counterpart v' in the signature, and such that v does not occur in S, we have $v' = v$.

A variant of the *changes_only* notation allows the use of a schema as an argument. If S is a schema, then $changes\_only(S)$ is equivalent to $changes\_only(V)$ where V is the set of variables in the signature (see Section 2.3.9) of S.

Note *changes_only* is *always* interpreted relative to the signature of the schema in which it appears. Therefore, when one schema is included in another, the instance of *changes_only* appearing in the first schema asserts nothing about the variables of the second.

A special case is $changes\_only\{\}$, which asserts that the operation changes the values of no state variables.

Example

op newuser

$users, users' : \mathbb{P}\, USER$

$books, books' : \mathbb{P}\, BOOK$

$new? : USER$

$users' = users \cup \{new?\}$

$changes\_only\{users\}$

```
op schema newuser is
dec
  users,users': power USER;
  books,books': power BOOK;
  new?  : USER
pred
  users' = users union new?;
  changes_only users
end newuser
```

In this schema, $changes\_only\{users\}$ is equivalent to $books = books'$.

3.2.3 Membership and Equality

Syntax

Expression $\in$ *Expression*	
Expression `in` *Expression*	– Membership
Expression $=$ *Expression*	
Expression `=` *Expression*	– Equality

Description The predicate $e_1 \in e_2$ is true if e_2 is a set, and e_1 is a member of that set.

The predicate $e_1 = e_2$ is true if e_1 and e_2 both have the same value (as sets, if they both have the same members) and false otherwise.

Examples

$1 \in \{1, 2, 3\}$	`1 in { 1,2,3 }`
$\{ x : \mathbb{N} \mid x < 4 \} = \{0, 1, 2, 3\}$	`{ dec x:nat | x < 4 } = { 0,1,2,3 }`

3.2.4 Infix and Prefix Relational Operators

Syntax

ExpressionInRelOpExpression	– infix relation
PreRelOpExpression	– prefix relation

Description The toolkit (Chapter 5) defines various infix relations. For such a relation $(\_ \Box \_)$, the expression $a \Box b$ is a shorthand for writing $(a, b) \in (\_ \Box \_)$.

Likewise, the toolkit may define a prefix relation, $P\_$. Writing Px is a shorthand for $x \in (P\_)$.

User-defined infix and prefix relations are not permitted.

Examples

$\mathbb{N} \subseteq \mathbb{Z}$ is equivalent to $(\mathbb{N}, \mathbb{Z}) \in (\_ \subseteq \_)$
disjoint S is equivalent to $S \in (\text{disjoint}\ \_)$

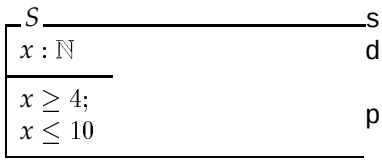
3.2.5 Schema (as predicate)

Syntax

CmpndSchema – schema reference

Description Any schema/schema expression may be used as a predicate. Such a schema denotes the predicate formed by asserting its signature as a membership predicate conjoined with its predicate part.

Example If the context contains a schema definition




```
schema S is
dec
  x:nat
pred
  x >= 4 ;
  x <= 10
end S
```

Then we have

$$S \Leftrightarrow (x \in \mathbb{N} \land x \geq 4 \land x \leq 10)$$

(Here *S* plays the role of a predicate.)

3.3 Procedural Constructs

Description Procedural constructs are modelled in the intermediate language as predicates relating before (unprimed) and after (primed) states. For further discussion on this see B.4.

As in various refinement calculi, Sum intermediate language constructs can be intermixed with more general 'non-IL' Sum specification constructs.

3.3.1 Multiple assignment

Syntax

$$\texttt{assign}\ x1, .., xn := E1, .., En$$

where

$$E1, .., En$$

are independent of any primed variables.

Description The meaning of the above multiple assignment is that

1. $x1' = E1 \wedge xn' = En$
2. For all variable pairs Y, Y′ in the current signature such that Y does not appear on the left of the assignment, Y′ = Y.

Note that all expressions on the right hand side of the assignment are evaluated before assignment.

Example

$$\texttt{assign}\ a, b, c := 1, 2, 3$$

3.3.2 If construct

Syntax

`ifc` *Predicate* `then` *Predicate* `fi`
`ifc` *Predicate* `then` *Predicate* `else` *Predicate* `fi`

Description In the construct

`ifc` $P1$ `then` $P2$

If the initial predicate, $P1$, is true, then the predicate $P2$ is achieved.

In the construct

`ifc` $P1$ `then` $P2$ `else` $P3$

If the initial predicate, $P1$, is true, then the predicate $P2$ is achieved. Otherwise, $P3$ is achieved.

Examples

```
ifc (alarm = ringing)
    then (assign order := evacuate)
fi
```

```
ifc (light = green)
    then (assign order := go)
else
    (assign order := stop)
fi
```

3.3.3 While construct

Syntax

`while` *Predicate* `do` *Predicate* `done`

Description In the construct

`while` $P1$ `do` $P2$ `done`,

in each iteration where $P1$ is true, $P2$ is achieved. Otherwise, the loop terminates.

Example

`while` $(index < 10)$ `do` $($`assign` $index := index + 1)$ `done`

3.3.4 Call construct

Syntax

`call` (*Name* , (), ())
`call` (*Name* , (*InputBindList*), ())
`call` (*Name* , (), (*OutputBindList*))
`call` (*Name* , (*InputBindList*), (*OutputBindList*))

Description This procedural construct represents procedure invocation, with or without parameters.

Input and output parameters have the form

Id => *Expression* {, *Id* => *Expression*}

where *Id* refers to a name declared in the function called and the *Expression* refers to locals known in the context of the *call*. In the case of the output parameters, *Expression* is restricted to a *Name*. This name is written unprimed, although the value returned is assigned to the primed variable.

Intermediate language function invocation uses the same syntax as Sum function invocation (see 4.11).

Example

`call` (*PrintRules* , () , ())

`call` (*BorrowBook*, (*name* => *BorrowersName*, *book* => *BookCatNo*), ())

`call` (*QueryPressure*, (), (*pressure* => *TankPressure*))

`call` (*GetAccountBalance*, (*no* => *AccountNo*),
(*balance* => *AccountBalance*))

3.3.5 Sequential composition

Syntax Predicate `;;` Predicate

Description The sequential composition of statements.

Example

```
assign index := 1 ;;
while keys (index ) ≠ key do
    assign index := index + 1 done
```

3.3.6 Blocks

Syntax

$$\texttt{var}\ \mathit{SchemaText} \bullet \mathit{Predicate}$$

Description The intermediate language quantifier `var` introduces a block. If there is any transition from pre to post states, e.g. assignment, on a quantified variable, both the primed and unprimed states of that variable must be explicitly declared in the *SchemaText*.

Example

$$\begin{array}{l}\texttt{var}\ i, i' : 1\ \texttt{uptoc}\ k \bullet \\ \quad \texttt{assign}\ i := 1;\ ; \\ \quad\quad \texttt{while}\ \mathit{nos}(i)\ \texttt{neqc}\ 10\ \texttt{do} \\ \quad\quad\quad \texttt{assign}\ i := i\ \texttt{addc}\ 1 \mathit{done}\end{array}$$

4. Expressions

4.1 Set Comprehension

Syntax

{ [dec] *SchemaText* [@ *Expression*] }

Description The set {*SchemaText* @ *Expression* } is the set of values taken by the *Expression* when the variables introduced by *SchemaText* take all possible values which make the property of *SchemaText* true.

The set {*SchemaText* } is the set of all tuples of values taken by the variables introduced in *SchemaText* such that the property of *SchemaText* is true.

When *SchemaText* is a single schema reference, S, the expression {dec S} is the set of all tuples of values taken by the variables introduced in S such that the property of S is true, while the expression $\{S\}$ is the set of the bindings of type S.

Examples

- The set of all even natural numbers less than 99

  $$\{ n : \mathbb{N} \mid \texttt{exists}\ m : \mathbb{N} @ 2m = n \ \ @ n < 99 \}$$

- The set of tuples (x, y) such that x and y are natural numbers, and each is less than 99:

  $$\{ x : \mathbb{N};\ y : \mathbb{N} \mid x < 99 \ \land y < 99\}$$

  This latter set is of type $\mathbb{P}(\mathbb{N} \times \mathbb{N})$, that is, it is a subset of $(\mathbb{N} \times \mathbb{N})$.

- The set of natural numbers less than 10:

  { dec S }

  where

  ```
  schema S is
  dec x : nat
  pred x < 10
  end S
  ```

- The set of bindings of type S:

  { S }

4.2 Set Enumeration

Syntax

{ [*ExpressionList*] }

Description
The set $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is the set containing only the members $\alpha_1, \alpha_2, \ldots, \alpha_n$.

Examples

$a \in \{a, b\}$
$c \notin \{a, b\}$

Laws

$\{\alpha_1, \alpha_2, \ldots, \alpha_n\}\downarrow \Rightarrow \exists\, t \bullet \alpha_1, \alpha_2, \ldots, \alpha_n : t$
$\neg\ \alpha_i\downarrow \wedge\ 1 \leq i \wedge i \leq n \Rightarrow \neg\ \{\alpha_1, \alpha_2, \ldots, \alpha_n\}\downarrow$
$\varnothing = \{\}$
$x \in \{e_1, e_2, \ldots, e_n\} \Leftrightarrow x = e_1 \vee x = e_2 \vee \ldots \vee x = e_n$

4.3 Cartesian Products

Syntax

Expression $\times \ldots \times$ *Expression*	
Expression `cross` ... *cross Expression*	– Cartesian product
`(` *Expression*`,` *ExpressionList* `)`	– *tuple*

Description
The cartesian product $(A \times B)$ is the set of all pairs where the first element is drawn from the set A and the second element is drawn from the set B. More generally, the cartesian product $A_1 \times \ldots \times A_n$ is the set of ordered tuples $(\alpha_1, \ldots, \alpha_n)$ where $\alpha_1 \in A_1, \ldots, \alpha_n \in A_n$.

Notice that $A \times B \times C$ and $A \times (B \times C)$ and $(A \times B) \times C$ are all distinct. Typical members of these are (a, b, c), $(a, (b, c))$, and $((a, b), c)$ respectively.

Examples

$\{a, b\} \times \{c, d\} = \{(a, c), (a, d), (b, c), (b, d)\}$
$(1, \text{true}) \in \mathbb{N} \times \mathbb{B}$

Laws

$(\alpha_1, \alpha_2, \ldots, \alpha_n)\downarrow \Rightarrow \exists\, t_1, t_2, \ldots, t_n \bullet (\alpha_1 : t_1 \wedge \alpha_2 : t_2 \wedge \ldots \wedge \alpha_n : t_n)$
$\neg\, \alpha_i\downarrow \wedge 1 \leq i \wedge i \leq n \Rightarrow \neg\, (\alpha_1, \alpha_2, \ldots, \alpha_n)\downarrow$

4.4 Power sets

Syntax

$\mathbb{P}$ *Expression*	`power`*Expression*

Description
The set $\mathbb{P}\,\alpha$ is the set of all subsets of the set α.

Example `power {a,b}={{},{a},{b},{a,b}}`

$$\mathbb{P}\{a, b\} = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$$

4.5 Tuple

Syntax

(*Expression* , *ExpList*)

Description The expression ($x1$, . ., xn) denotes an n-tuple whose components are $x1$, . ., xn. A tuple must have at least two expressions.

Example (2,4,6,8)

4.6 Characters

Syntax The character type:

char

Character literals:

'*Letter* | *Digit* | ! | @ | # | $ | ...'

Description The `char` type is a built-in type, denoting the set of characters.

Example

```
'a' in char
```

4.7 Strings

Syntax The string type:

string

String literals:

"[*Character*{*Character*}]"

Description The `string` type is a built-in type, denoting the set of characters.

Example

```
"abc" in string
```

4.8 Lambda Expressions

Syntax

λ *SchemaText* $\bullet$ *Expression* `lambda` *SchemaText* `@` *Expression*

Description The expression `lambda` *SchemaText* `@` E denotes a function which takes arguments which are tuples of type determined by *SchemaText* and returns the result E.

Examples

- $\lambda\, x : \mathbb{N};\ y : \mathbb{N} \mathbin{@} x + y$ denotes the function which returns for each tuple (x, y) in $\mathbb{N} \times \mathbb{N}$ the value resulting from adding x and y. Thus

  $$(3, 4) \mapsto 7 \in (\lambda\, x : \mathbb{N};\ y : \mathbb{N} \mathbin{@} x + y)$$

- $$\lambda\, S \mathbin{@} x$$

  takes tuples to x.

4.9 Mu Expressions

Syntax

$$\mu \ \textit{SchemaText} \bullet [\ \textit{Expression}] \qquad \texttt{mu}\ \textit{SchemaText}\ [\ @\ \textit{Expression}\ ]$$

Description The expression mu *SchemaText* @ *E* is defined only if there are unique values of the variables introduced in *SchemaText* which make the property of *SchemaText* true. If that is so, the value of the Mu expression is the value of *Expression* when the variables introduced by *SchemaText* take these values.

Examples

- $\mu\, x : \mathbb{N};\ y : \mathbb{N} \mid ((x = 3) \land (y = 4))$ denotes the tuple $(3, 4)$.
- $\mu\, x : \mathbb{N};\ y : \mathbb{N} \mid ((x = 3) \land (y = 4))\ @\ x + y$ has type $\mathbb{N}$ and value 7.

4.10 Theta expressions

Syntax

θ *SchemaText* `theta` *Name*

Description The expression `theta` S', denotes an anonymous schema, whose variables take their names from the schema S and their types and values from variables, of the same name and decorated as specified, in the current environment. All the decorated variables of S must be in scope.

Example Given

```
schema S is
dec x : nat
end S
```

the expression,

`lambda` S `@ theta` S

denotes the bindings $(x \mapsto v)$, where v is the value of x in the lambda expression.

`lambda` $S; S' \mid x' = x + 1 \bullet$ `theta` S'

takes pairs (v, v') satisfying $v' = v + 1$ and gives bindings $(x \mapsto v')$ where v' is the value taken by x' in the lambda expression.

4.11 Function application

Syntax

BasicExpression (*ExpressionList*)

Description The expression $f(E)$ denotes the result of applying the function f to E, provided that f is a function defined at E.

If E is not in the domain of f, or f is not functional at E (by mapping E to a number of different values), the result is the undefined expression.

4.12 If expressions

Syntax

`if` *Predicate* `then` *Expression* `else` *Expression* `fi`

Description The expression `if` P `then` $E1$ `else` $E2$ `fi` evaluates to $E1$, if P evaluates to true, and to $E2$ otherwise.

Example The expression `if` $1 > 0$ `then` 4 `else` 5 `fi` evaluates to 4.

4.13 Sequence enumeration

Syntax

⟨ [*ExpressionList*] ⟩ < [*ExpressionList*] >

Description
The sequence $\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$ is the sequence containing the members α_1, α_2, $\ldots$, α_n in that order.

Examples

$\langle a, b \rangle \frown \langle b, c \rangle = \langle a, b, b, c \rangle$

Laws

$\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle \downarrow \Rightarrow \exists\, t \bullet \alpha_1, \alpha_2, \ldots, \alpha_n : t$
$\neg\ \alpha_i \downarrow \wedge\ 1 \leq i \wedge i \leq n \Rightarrow \neg\ \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle \downarrow$

4.14 Bag enumeration

Syntax

⟦ [*ExpressionList*] ⟧ |[[*ExpressionList*]]|

Description
The bag $[\![\alpha_1, \alpha_2, \ldots, \alpha_n]\!]$ is the bag containing only the members $\alpha_1, \alpha_2, \ldots, \alpha_n$.

Examples

a in_bag $[\![a, b]\!]$
$[\![a, b]\!] \uplus [\![b, c]\!] = [\![a, b, b, c]\!]$

```
a in_bag |[a,b]|
|[a,b]| bag_union |[b,c]|
                    = |[a,b,b,c|]
```

Laws

$$[\![\alpha_1, \alpha_2, \ldots, \alpha_n]\!]\downarrow \Rightarrow \exists\, t \bullet \alpha_1, \alpha_2, \ldots, \alpha_n : t$$
$$\neg\ \alpha_i\downarrow \wedge 1 \leq i \wedge i \leq n \Rightarrow \neg\ [\![\alpha_1, \alpha_2, \ldots, \alpha_n]\!]\downarrow$$

4.15 Schema (as set/type)

Syntax

Expression

Description A schema expression (schema reference) can be used to denote a set of *bindings*. A binding is a (finite) mapping from schema variables to values. The set denoted by a schema S is the set of bindings which map each variable of S to a value of the the type specified in the signature of S .

The identifiers declared in the schema serve as selectors for *components* of the binding—see Section 4.16.

Example For example:

```
schema Date is
dec
   day: 1 .. 31;
   month: 1 .. 12;
   year: nat
pred
   is_valid_date(day,month,year)
end Date;
```

introduces a set (or type) `Date`.

A typical binding in `Date` may be shown as: (this is not a Sum expression)

$$\{\texttt{day} \mapsto 22, \texttt{month} \mapsto 12, \texttt{year} \mapsto 1994\}$$

Variables may be declared of date type: e.g.

```
start: Date
```

Elements of `date` satisfy the constraint which is given by the predicate part of the schema. e.g.

```
is_valid_date(start.day, start.month, start.year)
```

4.16 Binding Selection

Syntax

Name `.` *Id*

Description In the expression $b.x$, b must be a binding of schema type (see 4.15) having x as a component. The expression denotes the value of the x component of the binding.

Example Let S be the schema

```
schema S is
dec
  x,y:nat
pred
x>y
end S
```

S

$$x, y : \mathbb{N}$$

$$x > y$$

Let $b : S$. Then $b.x$ denotes the x component of b and $b.y$ denotes the y component of b.

4.17 Schema Expressions

Schemas are introduced in section 2.3.9. This section describes a number of schema expressions.

4.17.1 Logical Schema Operators

Syntax Let *opr* stand for one of the logical operators $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ (`and`, `or`, `not` `=>`, `<=>`). The syntax for a logical schema expression using the operator *opr*

CmpndSchema opr CmpndSchema

Description The *schema or* operator, for example, produces a schema such that

- the signature part is the merging of the signature parts of the arguments, and
- the predicate part is the logical disjunction of the predicate parts of the arguments.

Note that the result is well-defined only when the two schemas are signature compatible, that is, the declared sets of each variable common to the signatures of both schemas are equal.

Thus, in general, if `S` and `T` are schemas and `opr` is one of the operators, then `S` *opr* `T` forms a schema from `S` and `T` by merging their declarations (provided they are compatible), and combining their predicates using *opr*.

Example

```
schema S1 is
dec
  x,y:nat
pred
  x > y
end S1
```

```
schema S2 is
dec
 y,z:nat
pred
 y > z
end S2
```

Then `S1 or S2` is equivalent to

```
schema S1orS2 is
dec
  x,y,z:nat
pred
   (x > y) or (y > z)
end S1orS2
```

4.17.2 Schema Hiding

Syntax

CmpndSchema `\ (` *SchVarList* `)`

Description If `S` is a schema then `S \ (v1,v2, ...)` is a schema with variables `v1`, `v2`,... hidden. The variables listed are removed from the declarations and are existentially quantified in the predicate.

Example Let

```
schema S is
dec
   x:int;
   y:power int
pred
   x <= #y
end S
```

Then `S \ (x)` is the schema

```
schema S_hide is
dec
   y:power int
pred
   exists x:int @ x <= #y
end S_hide
```

4.17.3 Schema Projection

Syntax

CmpndSchema `|^` *BasicSchema*

Description If `S` and `T` are schemas with compatible signatures then `S |^T` is a schema equivalent to `(S and T)\(v1,v2, ..., vn)` where `v1,v2, ..., vn` are the components of `S` not also in `T`.

4.17.4 Schema Composition

Syntax

CmpndSchema `s_compose` *CmpndSchema*
CmpndSchema $\fatsemi$ *CmpndSchema*

Description The schema composition `S1 s_compose T1` represents the sequential composition of `S1` and `T1` by identifying the 'after state' of `S1` (i.e. its dashed variables) with the 'before state' of `T1` (corresponding undashed variables) and hiding this intermediate state.

To form the composition `S1 s_compose T1` the pairs of after-state components of `S1` and before-state components of `T1` that have the same base name are renamed to a new variable, the resulting schemas are conjoined, and the new variables hidden.

Let `x1':Ty1`, ..., `xn':Tyn` be the dashed variables of S1 which have corresponding undashed variables `x1:Ty1`, ..., `xn:Tyn` in `T1`. Then `S1 s_compose T1` equals

```
exists v1:Ty1; ... ; vn:Tyn @

(S1{v1/x1', ..., vn/xn'} and
T1{v1/x1, ..., vn/xn})
```

Example Let

```
schema S1 is
dec
   x?,s,s',y!:nat
pred
   s' = s - x? and y! = s'
end S;

schema T1 is
dec
   x?,s,s':nat
pred
   s <= x? and s' = s + x?
end T
```

Then `S1 s_compose T1` is the schema

```
schema S1_cmp_T1 is
dec
   x?,s,s',y!:nat
pred
   exists ss:nat @
     ss = s - x? and y! = ss
     and ss <= x? and s' = ss + x?
end S1_cmp_T1
```

This simplifies to

```
schema S1_cmp_T1 is
dec
   x?,s,s',y!:nat
pred
   y! = s - x? and s - x ? <= x? and s' = s
end S1_cmp_T1
```

4.17.5 Schema Renaming

Syntax

CmpndSchema {*DeclName*/*Name* {, *DeclName*/*Name*}}

Description The schema `S{new1/old1,new2/old2, ...}` is the schema `S` with variable `old1` renamed to `new1`, `old2` to `new2` etc.

The renaming occurs in both the declaration and the predicate part. If the renaming leads to two or more previously distinct variables ending up with the same name, the renaming is valid only if the variables all have the same type. Renaming in the predicate may entail renaming of bound variables.

4.17.6 Schema Inclusion

Description A schema can be included in the declaration part of another schema. The effect is that the included schema has its declaration added to the new schema, and its predicate added (by `and`) to the predicate of the new schema.

Example Let

```
schema deltabuffer is
dec
   items,items': seq MSG
pred
   #items <= maximum;
   #items' <= maximum
end delta_buffer
```

Now define `join` by including this schema:

```
schema join is
dec
   deltabuffer; msg?:MSG
pred
   #items < maximum;
   items' = itemsˆ<msg?>
end join
```

This is equivalent to

```
schema join is
dec
   items, items': seq MSG;
   msg?: MSG
pred
   #items <= maximum;
   #items' <= maximum;
   #items < maximum;
   items' = items ˆ <msg?>
end join;
```

4.17.7 Schema decoration

Description Decorating a schema S results in a schema derived from S by systematically decorating each of the components of S. Typically the decoration is a prime ($'$); therefore forming the schema S' entails adding a prime to each of the components of S throughout the signature and predicate parts of S.

Example Let the schema `buffer_state` be defined by

```
schema buffer_state is
dec
   items: seq MSG
pred
   #items <= maximum
end buffer_state
```

Now `buffer_state'` has its identifiers systematically dashed as follows:

```
schema buffer_state' is
dec
   items': seq MSG
pred
   #items' <= maximum
end buffer_state
```

4.18 Intermediate language expressions

These are part of the intermediate language subset and can be translated directly to the target programming language (see B.4).

4.18.1 Array component update

Syntax

`upd` (*Name* , *Expression* , *Expression*)

Description Array update is represented by a restricted version of function override which does not extend its domain (since this is not possible for arrays). The expression

`upd` (a , i , v)

denotes the function (array) a updated at index i to the value v .
Array types are described in B.4.3

Example

keep : `array` (`nat` , `nat`)
high : `nat`
$keep' = upd$ (*keys* , $high + 1$, 311)

4.18.2 Array expression list

Syntax

(*Expression* `is` *Expression* { , *Expression* `is` *Expression* })

Description This expression is really a parenthesised mapping or tuple of mappings. It allows the "use of" the functional representation of arrays when referring to individual elements and their values. However, the infix operator, `is` , is used to distinguish the expression, as pertaining to arrays, at the top level.

Example

ab : `enum` (*a*, *b*)
pairs : `array` (*ab*, `nat`)
pairs′ = (*a* `is` 3 , *b* `is` 6)

4.18.3 Record display

Syntax

`the` *SchemaText* `@` (*AndPredList*)

Description This unique choice operator can be used to represent record aggregates. The expression

`the` *SchemaText* `@` *AndPredList*

is defined only if there is a unique way of giving values to the variables of *SchemaText* which make the property of *AndPredList* true.

Example A target language record type for date can be represented in Sum by

```
schema date is
dec
    day : 1..31;
    month : month_name;
    uear : nat
end date
```

The record aggregate

$$(day \Rightarrow 17,\ month \Rightarrow mar,\ year \Rightarrow 99\ )$$

can be represented by the expression

$$\texttt{the}\ d : date\ \texttt{@}\ (d.day = 17\ \texttt{and}\ d.month = mar\ \texttt{and}\ d.year = 99\ )$$

of schema type date.

5. Toolkit

5.1 Arithmetic

5.1.1 Integers and Subsets of Integers

Names

$\mathbb{Z}$	`int`	– The set of integers
$\mathbb{N}$	`nat`	– The set of natural numbers
$\mathbb{N}_1$	`nat_1`	– The set of non-zero natural numbers
$\_\ldots\_$	`_..._`	– Set of consecutive integers constructor

Description
These are the usual subsets of integers.

Examples

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

Definitions

$\mathbb{N} == \{n : \mathbb{Z} \mid 0 \leq n\}$
$\mathbb{N}_1 == \{n : \mathbb{N} \mid 0 < n\}$

$$\begin{array}{|l} x..y : \mathbb{Z} \times \mathbb{Z} \to \mathbb{P}\,\mathbb{Z} \\ \hline x..y = \{a : \mathbb{Z} \mid x \leq a \leq y\} \end{array}$$

Laws

$\mathbb{Z}\downarrow$
$\mathbb{N}\downarrow$
$\mathbb{N}_1\downarrow$
$\forall x, y : \mathbb{Z} \bullet$
$\quad (x..y)\downarrow$

5.1.2 Arithmetic Functions

Names

$\_ + \_$	`_+_`	– Integer addition
$\_ - \_$	`_-_`	– Integer subtraction
$\_ * \_$	`_*_`	– Integer multiplication
*_div_*	`_div_`	– Integer division
*_mod_*	`_mod_`	– Integer remainder on division
*succ_*	`succ_`	– Successor of an integer
*min_*	`min_`	– Minimum of a set of integers
*max_*	`max_`	– Maximum of a set of integers
$\_ * * \_$	`_**_`	– Integer exponentiation

Description
These operators are the usual operators on integers.

Examples

$2 + 2 = 4$
$2 - 1 = 1$
$2 * 3 = 6$
$2 * *3 = 8$
5 *div* $3 = 1$
5 *mod* $3 = 2$
succ$(1) = 2$
min$\{3, -1, 2, 1\} = -1$
max$\{-1, 2, 3, 1\} = 3$

Laws

$$
\begin{array}{l}
succ(x)\downarrow \Leftrightarrow x : \mathbb{Z} \\
(x + y)\downarrow \Leftrightarrow x, y : \mathbb{Z} \\
(x - y)\downarrow \Leftrightarrow x, y : \mathbb{Z} \\
(x * y)\downarrow \Leftrightarrow x, y : \mathbb{Z} \\
(x * *y)\downarrow \Leftrightarrow x : \mathbb{Z} y : nat \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad \neg\ (y = 0) \Rightarrow (x \operatorname{div} y)\downarrow \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad \neg\ (y = 0) \Rightarrow (x \bmod y)\downarrow \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad x * succ(y) = x * y + x \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad x + succ(y) = succ(x + y) \\
\forall\, x : \mathbb{Z} \bullet \\
\quad -x + x = 0 \\
\forall\, x, y, z : \mathbb{Z} \bullet \\
\quad x + (y + z) = (x + y) + z \\
\forall\, x : \mathbb{Z} \bullet \\
\quad 0 + x = x \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad x + y = y + x \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad x, y : \mathbb{Z} (x * y = 0 \Leftrightarrow x = 0 \text{ or } y = 0) \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad w * *(x + y) = w * *x * w * *y \\
\forall\, x, y, z : \mathbb{Z} \bullet \\
\quad x * (y * z) = (x * y) * z \\
\forall\, x, y, z : \mathbb{Z} \bullet \\
\quad (x + y) * z = x * z + y * z \\
\forall\, x, y, w, z : \mathbb{Z} \bullet \\
\quad x = y \land w = z \Rightarrow z * w = y * z \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad -x * y = x * -y \\
\forall\, x : \mathbb{Z} \bullet \\
\quad 1 * x = x \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad x * y = y * x \\
\forall\, x : \mathbb{Z} \bullet \\
\quad 0 * x = 0 \\
\forall\, x : \mathbb{Z} \bullet \\
\quad -x = -1 * x \\
\forall\, x, y : \mathbb{Z} \bullet \\
\quad succ(x + y) = succ(x) + y
\end{array}
$$

5.1.3 Arithmetic Relations

Names

$\_ > \_$	`_>_`	– Integer greater than
$\_ < \_$	`_<_`	– Integer less than
$\_ \geq \_$	`_>=_`	– Integer greater than or equal to
$\_ \leq \_$	`_<=_`	– Integer less than or equal to

Description
These operators are the usual boolean relations on integers.

Examples

$1 > 0$
$-1 < 2$
$1 \geq 1$
$-1 \leq 0$

Laws

$$(x < y)\downarrow \Leftrightarrow x, y : \mathbb{Z}$$
$$(x \leq y)\downarrow \Leftrightarrow x, y : \mathbb{Z}$$
$$(x > y)\downarrow \Leftrightarrow x, y : \mathbb{Z}$$
$$(x \geq y)\downarrow \Leftrightarrow x, y : \mathbb{Z}$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x < y \Leftrightarrow \exists w : \mathbb{N}_1 x + w = y$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x \leq y \Leftrightarrow x < y \text{ or } x = y$$
$$\forall x \mathbb{Z} \bullet$$
$$\quad x \leq x$$
$$\forall x, y, z : \mathbb{Z} \bullet$$
$$\quad x \leq y \land y \leq z \Rightarrow x \leq z$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x \leq y \land y \leq z \Leftrightarrow x = y$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x \leq y \text{ or } y \leq x$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x \leq y \land y \leq succ(x) \Rightarrow x = y \text{ or } y = succ(x)$$
$$\forall x \mathbb{Z} \bullet$$
$$\quad x < succ(x)$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x \leq y \Leftrightarrow \neg\, y < x$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x = y \Rightarrow \neg\, x \leq y$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x = y \text{ or } x < y \text{ or } y < x$$
$$\forall x, y, z : \mathbb{Z} \bullet$$
$$\quad x < y \land y \leq z \Rightarrow x < z$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x < y \Leftrightarrow \neg\, y < x$$
$$\forall x, y, z : \mathbb{Z} \bullet$$
$$\quad x < y \land y < z \Rightarrow x < z$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad 0 \leq x \land 0 \leq y \Rightarrow 0 \leq x + y$$
$$\forall x, y, z : \mathbb{Z} \bullet$$
$$\quad x + y < z + y \Leftrightarrow x < z$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x + (-y) < 0 \Leftrightarrow x < y$$
$$\forall x : \mathbb{Z} \bullet$$
$$\quad -x < 0 \Leftrightarrow 0 < x$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad 0 \leq x \land 0 \leq y \Rightarrow 0 \leq x * y$$
$$\forall x, y : \mathbb{Z} \bullet$$
$$\quad x < y \Leftrightarrow x + 1 \leq y$$

5.2 Set Theory

Sum's set theory is a typed version of classical set theory. Good general references for set theory include [4, 10, 20].

5.2.1 Empty Set

Name

$\varnothing$ $\quad \{\ \}$ – Empty set

Description
The empty set (i.e. the set with no members).

Examples

$\varnothing = \{\}$

Definition

$$[X]$$
$$\varnothing : \mathbb{P}\, X$$
$$\varnothing = \{x : X \mid \text{false}\}$$

Laws

$\varnothing\downarrow$

$\varnothing = \{\}$

$\forall x : X \bullet$
$\quad x \notin \varnothing$

$\forall x : \mathbb{P}\, X \bullet$
$\quad x \cup \varnothing = x \land$
$\quad x \cap \varnothing = \varnothing$

$\forall x : \mathbb{P}\, X \bullet$
$\quad \varnothing \subseteq x$

5.2.2 Set Membership Relations

Name

$\_ \notin \_$ $\quad$ `_not_in_` – Set nonmembership

Description
The relation $\alpha \notin \beta$ means that the object α is *not* a member of the set β.

Examples

$a \notin \{b, c\}$

Definition

$[X]$

$$\_ \notin \_ : X \leftrightarrow \mathbb{P}\, X$$

$$\forall x : X;\ y : \mathbb{P}\, X \bullet \\ \quad x \notin y \Leftrightarrow \neg\, x \in y$$

Laws

$$(\alpha \notin \beta)\downarrow \Leftrightarrow \exists t \bullet (\alpha : t \land \beta : \mathbb{P}\, t)$$

$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \notin \beta)\downarrow$$

$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \notin \beta)\downarrow$$

$$\forall x : X \bullet \\ \quad x \notin \varnothing$$

$$\forall x : X;\ y, z : \mathbb{P}\, X \bullet \\ \quad x \in (y \setminus z) \Leftrightarrow x \in y \land x \notin z$$

5.2.3 Subset Relations

Name

$\_ \subseteq \_$	`_subset_`	– Subset relation
$\_ \subset \_$	`_p_subset_`	– Proper subset relation

Description
The relation $\alpha \subseteq \beta$ means that all the members of the set α are also members of the set β. The relation $\alpha \subset \beta$ means that all the members of the set α are also members of the set β but $\alpha \neq \beta$.

Examples

$\{a, b\} \subseteq \{a, b, c\}$
$\{a, b\} \subset \{a, b, c\}$
$\{a, b, c\} \subseteq \{a, b, c\}$
$\neg\ (\{a, b, c\} \subset \{a, b, c\})$

Definition

$[X]$

$\_ \subseteq \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X$
$\_ \subset \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X$

$\forall\, x, y : \mathbb{P}\, X \bullet$
$\quad x \subseteq y \Leftrightarrow \forall\, z : X \bullet (z \in x \Rightarrow z \in y)$
$\forall\, x, y : \mathbb{P}\, X \bullet$
$\quad x \subset y \Leftrightarrow x \subseteq y \land x \neq y$

Laws

$(\alpha \subseteq \beta)\downarrow \Leftrightarrow \exists\, t \bullet (\alpha : \mathbb{P}\, t \land \beta : \mathbb{P}\, t)$
$(\alpha \subset \beta)\downarrow \Leftrightarrow \exists\, t \bullet (\alpha : \mathbb{P}\, t \land \beta : \mathbb{P}\, t)$
$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \subseteq \beta)\downarrow$
$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \subseteq \beta)\downarrow$
$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \subset \beta)\downarrow$
$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \subset \beta)\downarrow$
$\forall\, x : \mathbb{P}\, X \bullet$
$\quad x \subseteq x \land$
$\quad \neg\ (x \subset x)$
$\forall\, x, y, z : \mathbb{P}\, X \bullet$
$\quad x \subseteq y \land y \subseteq z \Rightarrow x \subseteq z \land$
$\quad x \subset y \land y \subset z \Rightarrow x \subset z \land$
$\quad x \subseteq y \land y \subset z \Rightarrow x \subset z \land$
$\quad x \subset y \land y \subseteq z \Rightarrow x \subset z$
$\forall\, x, y : \mathbb{P}\, X \bullet$
$\quad x \subseteq y \land y \subseteq x \Leftrightarrow x = y$
$\forall\, x : \mathbb{P}\, X \bullet$
$\quad \varnothing \subseteq x$

5.2.4 Power Set Operators

Name

$\mathbb{P}_1$ _ `power_1_` – Nonempty power set

Description
The set $\mathbb{P}_1\,\alpha$ is the set of all nonempty subsets of the set α.

Examples

$\mathbb{P}_1\{a, b\} = \{\{a\}, \{b\}, \{a, b\}\}$

Definition

$\mathbb{P}_1\,X = \mathbb{P}\,X \setminus \{\varnothing\}$

Laws

$(\mathbb{P}_1\,\alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : \mathbb{P}\,t$
$\neg\ \alpha\downarrow \Rightarrow \neg\ (\mathbb{P}_1\,\alpha)\downarrow$
$\mathbb{P}_1\,\varnothing = \varnothing$
$\forall\, x : \mathbb{P}\,X \bullet$
$\quad \varnothing \notin \mathbb{P}_1\,x$

5.2.5 Intersection and Union Operators

Name

$\bigcup\_$	`gen_union_`	– Generalized (distributed) union
$\_\cup\_$	`_union_`	– Set union
$\bigcap\_$	`gen_inter_`	– Generalized (distributed) intersection
$\_\cap\_$	`_inter_`	– Set intersection

Description

The set $\bigcup \alpha$ is all the sets in the set α merged into a single set. The set $\alpha \cup \beta$ is the set of members of either of the sets α or β. The set $\bigcap \alpha$ is all the sets in the set α merged into a single set containing only the members of every set in α. The set $\alpha \cap \beta$ is the set of members of both of the sets α and β.

Examples

$$\bigcup \{\{a,b\},\{b,\{c\}\},\{d\}\} = \{a,b,\{c\},d\}$$
$$\{a,b\} \cup \{b,\{c\}\} = \{a,b,\{c\}\}$$
$$\bigcap \{\{a,b\},\{b,\{c\}\},\{b,\{c\},d\}\} = \{b\}$$
$$\{a,b\} \cap \{b,\{c\}\}\} = \{b\}$$

Definition

$[X]$

$$\bigcup\_ : \mathbb{P}\,\mathbb{P}\,X \to \mathbb{P}\,X$$
$$\_\cup\_ : \mathbb{P}\,X \times \mathbb{P}\,X \to \mathbb{P}\,X$$
$$\bigcap\_ : \mathbb{P}\,\mathbb{P}\,X \to \mathbb{P}\,X$$
$$\_\cap\_ : \mathbb{P}\,X \times \mathbb{P}\,X \to \mathbb{P}\,X$$

$$\forall x : \mathbb{P}\,\mathbb{P}\,X \bullet \bigcup x = \{y : X \mid \exists z : x \bullet y \in z\}$$
$$\forall x : \mathbb{P}\,\mathbb{P}\,X \bullet \bigcap x = \{y : X \mid \forall z : x \bullet y \in z\}$$
$$\forall x, y : \mathbb{P}\,X \bullet x \cup y = \bigcup\{x,y\}$$
$$\forall x, y : \mathbb{P}\,X \bullet x \cap y = \bigcap\{x,y\}$$

Laws

$$(\bigcup \alpha)\downarrow \Leftrightarrow \exists t \bullet \alpha : \mathbb{P}\,t$$
$$(\alpha \cup \beta)\downarrow \Leftrightarrow \exists t \bullet (\alpha : \mathbb{P}\,t \wedge \beta : \mathbb{P}\,t)$$
$$(\bigcap X)\downarrow \Leftrightarrow \exists t \bullet \alpha : \mathbb{P}\,t$$
$$(\alpha \cap \beta)\downarrow \Leftrightarrow \exists t \bullet (\alpha : \mathbb{P}\,t \wedge \beta : \mathbb{P}\,t)$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\bigcup \alpha)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \cup \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \cup \beta)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\bigcap \alpha)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \cap \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \cap \beta)\downarrow$$
$$\forall x, y : \mathbb{P}\,X \bullet x \cup y = y \cup x \wedge x \cap y = y \cap x$$

$$\forall x, y, z : \mathbb{P} X \bullet \\ x \cup (y \cup z) = (x \cup y) \cup z \land \\ x \cap (y \cap z) = (x \cap y) \cap z$$

$$\forall x, y, z : \mathbb{P} X \bullet \\ x \cup (y \cup z) = (x \cup y) \cup (x \cup z) \land \\ x \cap (y \cap z) = (x \cap y) \cap (x \cap z)$$

$$\forall x, y, z : \mathbb{P} X \bullet \\ x \cup (y \cap z) = (x \cup y) \cap (x \cup z) \land \\ x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$$

$$\forall x : \mathbb{P} X \bullet \\ x \cup x = x \land \\ x \cap x = x$$

$$\forall x : \mathbb{P} X \bullet \\ x \cup \varnothing = x \land \\ x \cap \varnothing = \varnothing$$

$$\forall x, y : \mathbb{P} X \bullet \\ x \cup y = \{z : X \mid z \in x \lor z \in y\}$$

$$\forall x, y : \mathbb{P} X \bullet \\ x \cap y = \{z : X \mid z \in x \land z \in y\}$$

$$\forall x : X;\ y : \mathbb{P} X;\ z : \mathbb{P}\mathbb{P} X \bullet \\ x \in y \land y \in z \Rightarrow x \in \bigcup z \land \\ x \in \bigcap z \land y \in z \Rightarrow x \in y$$

$$\forall x : X;\ y, z : \mathbb{P} X \bullet \\ x \in y \Rightarrow x \in (y \cup z) \land \\ x \in (y \cap z) \Rightarrow x \in y$$

$$\forall x, y : \mathbb{P} X \bullet \\ x \subseteq x \cup y \land \\ x \cap y \subseteq x$$

$$\forall x, y, z : \mathbb{P} X \bullet \\ x \subseteq z \land y \subseteq z \Rightarrow (x \cup y) \subseteq z \land \\ x \subseteq z \land y \subseteq z \Rightarrow (x \cap y) \subseteq z$$

$$\forall x, y, z : \mathbb{P} X \bullet \\ x \subseteq y \lor x \subseteq z \Rightarrow x \subseteq (y \cup z) \land \\ x \subseteq y \land x \subseteq z \Rightarrow x \subseteq (y \cap z)$$

$$\forall x, y, z : \mathbb{P} X \bullet \\ x \subseteq (y \cap z) \Rightarrow x \subseteq (y \cup z)$$

$$\forall x, y : \mathbb{P} X \bullet \\ \bigcup (x \cup y) = (\bigcup x) \cup (\bigcup y)$$

$$\forall x, y : \mathbb{P} X \bullet \\ x \neq \varnothing \land y \neq \varnothing \Rightarrow \bigcap (x \cup y) = (\bigcap x) \cap (\bigcap y)$$

$$\forall x : X;\ y, z : \mathbb{P} X \bullet \\ x \in (y \cap z) \Rightarrow x \in y$$

$$\forall x, y : \mathbb{P} X \bullet \\ x \cap y \subseteq x \cup y$$

5.2.6 Set Difference Operators

Name

$\_ \setminus \_$	`_ diff _`	– Set difference
$\_ \sim \_$	`_ sym_diff _`	– Symmetric set difference

Description
The set $\alpha \setminus \beta$ is the set of members of the set α not in the set β. The set $\alpha \sim \beta$ is the set of members of exactly one of the sets α and β.

Examples

$\{a, b\} \setminus \{b, c\} = \{a\}$
$\{a, b\} \sim \{b, c\} = \{a, c\}$

Definition

$[X]$

$\_ \setminus \_ : \mathbb{P} X \times \mathbb{P} X \to \mathbb{P} X$
$\_ \sim \_ : \mathbb{P} X \times \mathbb{P} X \to \mathbb{P} X$

$\forall x, y : \mathbb{P} X \bullet$
$\quad x \setminus y = \{z : \mathbb{P} X \mid z \in x \land z \notin y\}$
$\forall x : \mathbb{P} X \bullet$
$\quad x \sim y = (x \setminus y) \cup (y \setminus x)$

Laws

$(\alpha \setminus \beta)\downarrow \Leftrightarrow \exists t \bullet (\alpha : \mathbb{P} t \land \beta : \mathbb{P} t)$
$(\alpha \sim \beta)\downarrow \Leftrightarrow \exists t \bullet (\alpha : \mathbb{P} t \land \beta : \mathbb{P} t)$
$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \setminus \beta)\downarrow$
$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \setminus \beta)\downarrow$
$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \sim \beta)\downarrow$
$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \sim \beta)\downarrow$

$\forall x, y : \mathbb{P} X \bullet$
$\quad x \sim y = y \sim x$

$\forall x, y, z : \mathbb{P} X \bullet$
$\quad x \sim (y \sim z) = (x \sim y) \sim z$

$\forall x : \mathbb{P} X \bullet$
$\quad x \setminus x = \varnothing \land$
$\quad x \sim x = \varnothing$

$\forall x : \mathbb{P} X \bullet$
$\quad x \setminus \varnothing = x \land$
$\quad \varnothing \setminus x = \varnothing \land$
$\quad \varnothing \sim x = x$

$\forall x, y : \mathbb{P} X \bullet$
$\quad (x \setminus y) \subseteq x \land$
$\quad (x \sim y) \subseteq (x \cup y)$

$\forall x, y : \mathbb{P} X \bullet$
$\quad (x \cap y) = \varnothing \Leftrightarrow x \setminus y = x \land$
$\quad (x \cap y) = \varnothing \Leftrightarrow x \sim y = x \cup y$

$\forall x, y, z : \mathbb{P} X \bullet$
$\quad (x \cup y) \setminus z = (x \setminus z) \cup (y \setminus z) \land$
$\quad x \setminus (y \cap z) = (x \setminus y) \cup (x \setminus z)$

$$\forall x, y, z : \mathbb{P}\, X \bullet$$
$$x \cup (y \setminus z) = (x \cup y) \setminus (z \setminus x) \land$$
$$x \cap (y \setminus z) = (x \cap y) \setminus z$$
$$\forall x, y : \mathbb{P}\, X \bullet$$
$$x \cap (y \setminus x) = \varnothing$$
$$\forall x, y : \mathbb{P}\, X \bullet$$
$$x \sim y = (x \cup y) \setminus (x \cap y)$$

5.2.7 Ordered Pair Projection Operators

Name

first _	`first_`	– Ordered pair projection
second _	`second_`	– Ordered pair projection

Description
The projection *first* α is the first element of the ordered pair α. The projection *second* α is the second element of the ordered pair α.

Examples

$$first(a, b) = a$$
$$second(a, b) = b$$

Definition

$$[X, Y]$$
$$first\_ : X \times Y \to X$$
$$second\_ : X \times Y \to Y$$
$$\forall x : X;\ y : Y \bullet$$
$$first(x, y) = x \land$$
$$second(x, y) = y$$

Laws

$$(first\ \alpha)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : l \times r$$
$$(second\ \alpha)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : l \times r$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (first\ \alpha)\downarrow$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (second\ \alpha)\downarrow$$
$$\forall x : X \times Y \bullet$$
$$(first\ x, second\ x) = x$$

5.3 Relations

5.3.1 Relation Constructors

Name

$\_ \leftrightarrow \_$	_ <- -> _	– Binary relation
$\_ \mapsto \_$	_ | - -> _	– Mapping

Description
The set $\alpha \leftrightarrow \beta$ is the set of all binary relations between α and β (i.e. $\alpha \leftrightarrow \beta$ is an abbreviation for $\mathbb{P}(\alpha \times \beta)$). The expression $\alpha \mapsto \beta$ means that α relates to β (i.e. $\alpha \mapsto \beta$ is an abbreviation for the ordered pair (α, β)).

Examples

$$\{a \mapsto c, a \mapsto d\} \in \{a, b\} \leftrightarrow \{c, d\}$$

Definition

$$X \leftrightarrow Y == \mathbb{P}(X \times Y)$$
$$x \mapsto y == (x, y)$$

Laws

$$(\alpha \leftrightarrow \beta)\downarrow \Leftrightarrow \exists l, r \bullet (\alpha : \mathbb{P}\, l \land \beta : \mathbb{P}\, r)$$
$$(\alpha \mapsto \beta)\downarrow \Leftrightarrow \exists l, r \bullet (\alpha : l \land \beta : r)$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \leftrightarrow \beta)\downarrow$$
$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \leftrightarrow \beta)\downarrow$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \mapsto \beta)\downarrow$$
$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \mapsto \beta)\downarrow$$

5.3.2 Domain and Range Operators

Name

dom_	`dom_`	– Domain of a binary relation
ran_	`ran_`	– Range of a binary relation

Description
The set $\mathrm{dom}\,\alpha$ is the domain of the binary relation α (i.e. the set of all the first elements of the ordered pairs in α). The set $\mathrm{ran}\,\alpha$ is the range of the binary relation α (i.e. the set of all the second elements of the ordered pairs in α).

Examples

$\mathrm{dom}\{a \mapsto b, c \mapsto d\} = \{a, c\}$
$\mathrm{ran}\{a \mapsto b, c \mapsto d\} = \{b, d\}$

Definition

$$
\begin{array}{|l}
[X, Y] \\
\hline
\mathrm{dom}\_ : (X \leftrightarrow Y) \to \mathbb{P}\, X \\
\mathrm{ran}\_ : (X \leftrightarrow Y) \to \mathbb{P}\, Y \\
\hline
\forall\, r : X \leftrightarrow Y \bullet \\
\quad \mathrm{dom}\, r = \{x : X \mid \exists\, y : Y \bullet (x \mapsto y) \in r\} \land \\
\quad \mathrm{ran}\, r = \{y : Y \mid \exists\, x : X \bullet (x \mapsto y) \in r\} \\
\hline
\end{array}
$$

Laws

$$(\mathrm{dom}\,\alpha)\downarrow \Leftrightarrow \exists\, l, r \bullet \alpha : l \leftrightarrow r$$
$$(\mathrm{ran}\,\alpha)\downarrow \Leftrightarrow \exists\, l, r \bullet \alpha : l \leftrightarrow r$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\mathrm{dom}\,\alpha)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\mathrm{ran}\,\alpha)\downarrow$$
$$\mathrm{dom}\,\varnothing = \varnothing$$
$$\mathrm{ran}\,\varnothing = \varnothing$$
$$\forall\, r : X \leftrightarrow Y \bullet \quad \mathrm{dom}\, r \subseteq X \land \quad \mathrm{ran}\, r \subseteq Y$$
$$\forall\, r, s : X \leftrightarrow Y \bullet \quad \mathrm{dom}(r \cup s) = (\mathrm{dom}\, r) \cup (\mathrm{dom}\, s) \land \quad \mathrm{ran}(r \cup s) = (\mathrm{ran}\, r) \cup (\mathrm{ran}\, s)$$
$$\forall\, r, s : X \leftrightarrow Y \bullet \quad \mathrm{dom}(r \cap s) \subseteq (\mathrm{dom}\, r) \cap (\mathrm{dom}\, s) \land \quad \mathrm{ran}(r \cap s) \subseteq (\mathrm{ran}\, r) \cap (\mathrm{ran}\, s)$$

5.3.3 Composition Operators

Name

$\_ \fatsemi \_$	`_f_compose_`	– Forward composition
$\_ \circ \_$	`_b_compose_`	– Function Composition

Description
The binary relation $\alpha \fatsemi \beta$ is the binary relation formed by composing the binary relation β with the binary relation α. Thus if α contains $(\eta \mapsto \zeta)$ and β contains $(\zeta \mapsto \kappa)$ then $\alpha \fatsemi \beta$ will contain $(\eta \mapsto \kappa)$. The binary relation $\alpha \circ \beta$ is the binary relation formed by composing the binary relation α with the binary relation β. Thus if β contains $(\eta \mapsto \zeta)$ and α contains $(\zeta \mapsto \kappa)$ then $\alpha \circ \beta$ will contain $(\eta \mapsto \kappa)$.

Examples

$$\{a \mapsto c, a \mapsto d\} \fatsemi \{c \mapsto e, f \mapsto g\} = \{a \mapsto e\}$$
$$\{c \mapsto e, f \mapsto g\} \circ \{a \mapsto c, a \mapsto d\} = \{a \mapsto e\}$$

Definition

$[X, Y, Z]$

$$\_ \fatsemi \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \to (X \leftrightarrow Z)$$
$$\_ \circ \_ : (Y \leftrightarrow Z) \times (X \leftrightarrow Y) \to (X \leftrightarrow Z)$$

$$\begin{aligned}
&\forall r : X \leftrightarrow Y;\ s : Y \leftrightarrow Z \bullet \\
&\quad r \fatsemi s = \{x : X;\ z : Z \mid \\
&\qquad \exists y : Y \bullet \\
&\qquad\quad (x \mapsto y) \in r \land (y \mapsto z) \in s\} \land \\
&\quad s \circ r = r \fatsemi s
\end{aligned}$$

Laws

$$(\alpha \fatsemi \beta)\downarrow \Leftrightarrow \exists x, y, z \bullet (\alpha : x \leftrightarrow y \land \beta : y \leftrightarrow z)$$
$$(\alpha \circ \beta)\downarrow \Leftrightarrow \exists x, y, z \bullet (\alpha : y \leftrightarrow z \land \beta : x \leftrightarrow y)$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \fatsemi \beta)\downarrow$$
$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \fatsemi \beta)\downarrow$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \circ \beta)\downarrow$$
$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \circ \beta)\downarrow$$
$$\begin{aligned}
&\forall r : (U \leftrightarrow V) \bullet \\
&\quad (\varnothing \fatsemi r) = \varnothing \land \\
&\quad (r \fatsemi \varnothing) = \varnothing \land \\
&\quad (\varnothing \circ r) = \varnothing \land \\
&\quad (r \circ \varnothing) = \varnothing
\end{aligned}$$
$$\begin{aligned}
&\forall r : (U \leftrightarrow V);\ s : (V \leftrightarrow W);\ t : (W \leftrightarrow X) \bullet \\
&\quad r \fatsemi (s \fatsemi t) = (r \fatsemi s) \fatsemi t \land \\
&\quad t \circ (s \circ r) = (t \circ s) \circ r
\end{aligned}$$
$$\begin{aligned}
&\forall r : (U \nrightarrow V);\ s : (V \nrightarrow W);\ x : U \bullet \\
&\quad (r \fatsemi s)(x) = s(r(x)) \land \\
&\quad (s \circ r)(x) = s(r(x))
\end{aligned}$$

5.3.4 Restriction Operators

Name

$\_ \lhd \_$	`_dom_restrict_`	– Domain restriction
$\_ \rhd \_$	`_ran_restrict_`	– Range restriction
$\_ \lhd \_$	`_dom_subtract_`	– Domain subtraction
$\_ \rhd \_$	`_ran_subtract_`	– Range subtraction

Description
The binary relation $\alpha \lhd \beta$ is the binary relation formed by discarding from the domain of the binary relation β any elements *not* in the set α. Similarly, the binary relation $\alpha \rhd \beta$ is the binary relation formed by discarding from the range of the binary relation α any elements *not* in the set β. The binary relation $\alpha \lhd \beta$ is the binary relation formed by discarding from the domain of the binary relation β any elements in the set α. Similarly, the binary relation $\alpha \rhd \beta$ is the binary relation formed by discarding from the range of the binary relation α any elements in the set β.

Examples

$$\{a\} \lhd \{a \mapsto b, a \mapsto c, b \mapsto a\} = \{a \mapsto b, a \mapsto c\}$$
$$\{b \mapsto a, c \mapsto a, a \mapsto b\} \rhd \{a\} = \{b \mapsto a, c \mapsto a\}$$
$$\{a\} \lhd \{a \mapsto b, a \mapsto c, b \mapsto a\} = \{b \mapsto a\}$$
$$\{b \mapsto a, c \mapsto a, a \mapsto b\} \rhd \{a\} = \{a \mapsto b\}$$

Definition

$[X, Y]$

$$\_ \lhd \_ : (\mathbb{P}\, X) \times (X \leftrightarrow Y) \to (X \leftrightarrow Y)$$
$$\_ \rhd \_ : (X \leftrightarrow Y) \times (\mathbb{P}\, Y) \to (X \leftrightarrow Y)$$
$$\_ \lhd \_ : (\mathbb{P}\, X) \times (X \leftrightarrow Y) \to (X \leftrightarrow Y)$$
$$\_ \rhd \_ : (X \leftrightarrow Y) \times (\mathbb{P}\, Y) \to (X \leftrightarrow Y)$$

$$\forall s : \mathbb{P}\, X;\ r : X \leftrightarrow Y \bullet$$
$$s \lhd r = \{x : X;\ y : Y \mid x \in s \land (x \mapsto y) \in r\} \land$$
$$s \lhd r = \{x : X;\ y : Y \mid x \notin s \land (x \mapsto y) \in r\}$$
$$\forall s : \mathbb{P}\, Y;\ r : X \leftrightarrow Y \bullet$$
$$r \rhd s = \{x : X;\ y : Y \mid y \in s \land (x \mapsto y) \in r\} \land$$
$$r \rhd s = \{x : X;\ y : Y \mid y \notin s \land (x \mapsto y) \in r\}$$

Laws

$$(\alpha \lhd \beta)\downarrow \Leftrightarrow \exists\, l, r \bullet (\alpha : \mathbb{P}\, l \land \beta : l \leftrightarrow r)$$
$$(\alpha \rhd \beta)\downarrow \Leftrightarrow \exists\, l, r \bullet (\alpha : l \leftrightarrow r \land \beta : \mathbb{P}\, l)$$
$$(\alpha \lhd \beta)\downarrow \Leftrightarrow \exists\, l, r \bullet (\alpha : \mathbb{P}\, l \land \beta : l \leftrightarrow r)$$
$$(\alpha \rhd \beta)\downarrow \Leftrightarrow \exists\, l, r \bullet (\alpha : l \leftrightarrow r \land \beta : \mathbb{P}\, l)$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \lhd \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \lhd \beta)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \rhd \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \rhd \beta)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \lhd \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \lhd \beta)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \rhd \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \rhd \beta)\downarrow$$

$$
\begin{array}{l}
\forall s : \mathbb{P}\, X;\ r : X \leftrightarrow Y \bullet \\
\quad s \lhd r = \mathrm{id}\, s \fatsemi r \land \\
\quad s \lhd r = (s \times Y) \cap r \land \\
\quad s \lhdd r = (X \setminus s) \lhd r \\
\forall s : \mathbb{P}\, Y;\ r : X \leftrightarrow Y \bullet \\
\quad r \rhd s = r \fatsemi \mathrm{id}\, s \land \\
\quad r \rhd s = (X \times s) \cap r \land \\
\quad r \rhdd s = r \rhd (Y \setminus s) \\
\forall s : \mathbb{P}\, X;\ r : X \leftrightarrow Y \bullet \\
\quad \mathrm{dom}(s \lhd r) = s \cap \mathrm{dom}\, r \land \\
\quad s \lhd r \subseteq r \land \\
\quad s \lhd r \cup s \lhdd r = r \\
\forall s : \mathbb{P}\, Y;\ r : X \leftrightarrow Y \bullet \\
\quad \mathrm{ran}(r \rhd s) = s \cap \mathrm{ran}\, r \land \\
\quad r \rhd s \subseteq r \land \\
\quad r \rhd s \cup r \rhdd s = r \\
\forall s : \mathbb{P}\, X;\ t : \mathbb{P}\, Y;\ r : X \leftrightarrow Y \bullet \\
\quad (s \lhd r) \rhd t = s \lhd (r \rhd t) \land \\
\quad (s \lhdd r) \rhdd t = s \lhdd (r \rhdd t) \land \\
\quad (s \lhd r) \rhdd t = s \lhd (r \rhdd t) \land \\
\quad (s \lhdd r) \rhd t = s \lhdd (r \rhd t) \\
\forall s, t : \mathbb{P}\, X;\ r : X \leftrightarrow Y \bullet \\
\quad s \lhd (t \lhd r) = (s \cap t) \lhd r \land \\
\quad s \lhdd (t \lhdd r) = (s \cup t) \lhd r \\
\forall s, t : \mathbb{P}\, Y;\ r : X \leftrightarrow Y \bullet \\
\quad (r \rhd s) \rhd t = r \rhd (s \cap t) \land \\
\quad (r \rhdd s) \rhdd t = r \rhdd (s \cup t)
\end{array}
$$

5.3.5 Inverse Operators

Name

$\_^\sim$ `inverse(_)` – Inverse relation

Description
The binary relation $\alpha^\sim$ is the inverse of the relation α (i.e. if α contains $(\eta \mapsto \zeta)$ then $\alpha^\sim$ contains $(\zeta \mapsto \eta)$).

Examples

$$\{a \mapsto b, c \mapsto d\}^\sim = \{b \mapsto a, d \mapsto c\}$$

Definition

$[X, Y]$

$$\_^\sim : (X \leftrightarrow Y) \to (Y \leftrightarrow X)$$

$$\forall r : X \leftrightarrow Y \bullet r^\sim = \{y : Y;\ x : X \mid (x \mapsto y) \in r\}$$

Laws

$$(\alpha^\sim)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : l \leftrightarrow r$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha^\sim)\downarrow$$
$$\varnothing^\sim = \varnothing$$
$$\forall r : X \leftrightarrow Y \bullet r^{\sim\sim} = r \land \mathrm{dom}(r^\sim) = \mathrm{ran}\, r \land \mathrm{ran}(r^\sim) = \mathrm{dom}\, r$$
$$\forall r : X \leftrightarrow Y;\ t : Y \leftrightarrow Z \bullet (r \fatsemi t)^\sim = t^\sim \fatsemi r^\sim \land (t \circ r)^\sim = r^\sim \circ t^\sim$$

5.3.6 Relational Iterators

Name

$\mathrm{id}\_$	`id_`	– Identity relation
$\_^{-}$	`iter(_,_)`	– Iteration
$\_^{+}$	`t_closure(_)`	– Transitive closure
$\_^{*}$	`rt_closure(_)`	– Reflexive transitive closure

Description
The binary relation $\mathrm{id}\,\alpha$ is the identity relation over the set α (i.e. $\mathrm{id}\,\alpha$ only relates elements of α to themselves). The binary relation α^{κ} is the binary relation α applied to itself κ times (i.e. $\alpha \fatsemi \alpha \fatsemi \ldots \fatsemi \alpha$). The binary relation α^{+} is the transitive closure of the binary relation α. The binary relation α^{*} is the reflexive transitive closure of the binary relation α.

Examples

$\mathrm{id}\{a, b, c\} = \{a \mapsto a, b \mapsto b, c \mapsto c\}$
$r^{-1} = r^{\sim}$
$r^{0} = \mathrm{id}\,\mathrm{dom}\,r$
$r^{1} = r$
$\{a \mapsto b, b \mapsto c, d \mapsto e\}^{2} = \{a \mapsto c\}$
$\{a \mapsto b, b \mapsto c, d \mapsto e\}^{+} = \{a \mapsto b, a \mapsto c, b \mapsto c, d \mapsto e\}$
$\{a \mapsto b, b \mapsto c\}^{*} = \{a \mapsto a, a \mapsto b, a \mapsto c, b \mapsto b, b \mapsto c, c \mapsto c\}$

Definition

$[X]$

$$
\begin{array}{l}
\mathrm{id}\_ : \mathbb{P}\,X \to (X \leftrightarrow X) \\
\_^{-} : (X \leftrightarrow X) \times \mathbb{Z} \to (X \leftrightarrow X) \\
\_^{+} : (X \leftrightarrow X) \to (X \leftrightarrow X) \\
\_^{*} : (X \leftrightarrow X) \to (X \leftrightarrow X) \\
\hline
\forall\, s : X \bullet \\
\quad \mathrm{id}\,s = \{x : s \bullet (x \mapsto x)\} \\
\forall\, r : X \leftrightarrow X \bullet \\
\quad r^{0} = \mathrm{id}\,\mathrm{dom}\,r \land \\
\quad \forall\, k : \mathbb{N}_1 \bullet \\
\qquad r^{k} = r \fatsemi r^{k-1} \land \\
\qquad r^{-k} = r \fatsemi (r^{\sim})^{k} \\
\forall\, r : X \leftrightarrow X \bullet \\
\quad r^{+} = \bigcup\{k : \mathbb{N}_1 \bullet r^{k}\} \land \\
\quad r^{*} = \bigcup\{k : \mathbb{N} \bullet r^{k}\}
\end{array}
$$

Laws

$(\mathrm{id}\,\alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : t$
$(\alpha^{\kappa})\downarrow \Leftrightarrow \kappa : \mathbb{Z} \land \exists\, t \bullet \alpha : t \leftrightarrow t$
$(\alpha^{+})\downarrow \Leftrightarrow \exists\, t \bullet \alpha : t \leftrightarrow t$
$(\alpha^{*})\downarrow \Leftrightarrow \exists\, t \bullet \alpha : t \leftrightarrow t$
$\neg\, \alpha\downarrow \Rightarrow \neg\, (\mathrm{id}\,\alpha)\downarrow$
$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha^{\kappa})\downarrow$
$\neg\, \kappa\downarrow \Rightarrow \neg\, (\alpha^{\kappa})\downarrow$
$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha^{+})\downarrow$
$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha^{*})\downarrow$

$$\mathrm{id}\,\varnothing = \varnothing$$

$$\forall s : \mathbb{P} X \bullet \\ (\mathrm{id}\, s)^{\sim} = (\mathrm{id}\, s)$$

$$\forall r : X \leftrightarrow X \bullet \\ r^{-1} = r^{\sim} \land \\ r^0 = \mathrm{id}\,\mathrm{dom}\, r \land \\ r^1 = r \land \\ r^2 = r \fatsemi r$$

$$\forall r : X \leftrightarrow X;\ k : \mathbb{Z} \bullet \\ r^{k+1} = r^k \fatsemi r \land \\ r^{k+1} = r \fatsemi r^k$$

$$\forall r : X \leftrightarrow X;\ k : \mathbb{Z} \bullet \\ r^{\sim k} = r^{k\sim}$$

$$\forall r : X \leftrightarrow X;\ k, l : \mathbb{Z} \bullet \\ r^{k+l} = r^k \fatsemi r^l \land \\ r^{k*l} = r^{k^l}$$

$$\forall r : X \leftrightarrow X \bullet \\ r^{+ +} = r^+ \land \\ r^{**} = r^* \land \\ r^{+ *} = r^* \land \\ r^{* +} = r^*$$

$$\forall r : X \leftrightarrow X \bullet \\ r \subseteq r^+ \land \\ r \subseteq r^*$$

$$\forall r : X \leftrightarrow X \bullet \\ r^+ \fatsemi r^+ \subseteq r^+ \land \\ r^* \fatsemi r^* = r^*$$

$$\forall r : X \leftrightarrow X \bullet \\ r^+ = r \fatsemi r^* \land \\ r^+ = r^* \fatsemi r$$

$$\forall r : X \leftrightarrow X \bullet \\ r^* = r^+ \cup r^0 \land \\ r^* = (r \cup r^0)^+$$

$$\forall r : X \leftrightarrow X \bullet \\ r^0 \subseteq r^*$$

$$\forall r : X \leftrightarrow X \bullet \\ r^+ = \bigcap \{ x : \mathrm{dom}\, r \leftrightarrow \mathrm{dom}\, r \mid \\ \qquad r \subseteq x \land x \fatsemi x \subseteq x \} \land \\ r^* = \bigcap \{ x : \mathrm{dom}\, r \leftrightarrow \mathrm{dom}\, r \mid \\ \qquad r^0 \subseteq x \land r \subseteq x \land x \fatsemi x \subseteq x \}$$

5.4 Functions

5.4.1 Partial and Total Functions

Name

$\_ \pfun \_$	$\_$ - | -> $\_$	– Partial functions
$\_ \fun \_$	$\_$ - -> $\_$	– Total functions

Description
The set $\alpha \pfun \beta$ is the set of all functions from α to β. The set $\alpha \fun \beta$ is the set of all total functions from α to β.

Examples

$$\{a, b\} \pfun \{c\} = \{\{\}, \{(a \mapsto c)\}, \{(b \mapsto c)\}, \{(a \mapsto c), (b \mapsto c)\}\}$$
$$\{a, b\} \fun \{c\} = \{\{(a \mapsto c), (b \mapsto c)\}\}$$

Definition

$$X \pfun Y == \{f : X \rel Y \mid \forall u : X;\ v, w : Y \bullet (u, v) \in f \land (u, w) \in f \Rightarrow v = w\}$$
$$X \fun Y == \{f : X \pfun Y \mid \mathrm{dom} f = X\}$$

Laws

$$(\alpha \pfun \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$
$$(\alpha \fun \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \pfun \beta)\downarrow$$
$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \pfun \beta)\downarrow$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \fun \beta)\downarrow$$
$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \fun \beta)\downarrow$$
$$\forall f : X \pfun Y \bullet \mathrm{dom} f \subseteq X \land \mathrm{ran} f \subseteq Y$$
$$\forall f : X \fun Y \bullet \mathrm{dom} f = X \land \mathrm{ran} f \subseteq Y$$

5.4.2 Partial and Total Injections

Name

$\_ \rightarrowtail \_$ `_>-|->_` – Partial injections (one-to-one)
$\_ \rightarrowtail \_$ `_>->_` – Total injections (one-to-one)

Description
The set $\alpha \rightarrowtail \beta$ is the set of all injective (one-to-one) functions from α to β. The set $\alpha \rightarrowtail \beta$ is the set of all total injective (one-to-one) functions from α to β.

Examples

$$\{a, b\} \rightarrowtail \{c, d\} = \{\{\}, \{(a \mapsto c)\}, \{(b \mapsto c)\}, \{(a \mapsto c), (b \mapsto d)\}, \{(a \mapsto d)\}, \{(b \mapsto d)\}, \{(a \mapsto d), (b \mapsto c)\}\}$$
$$\{a, b\} \rightarrowtail \{c, d\} = \{\{(a \mapsto c), (b \mapsto d)\}, \{(a \mapsto d), (b \mapsto c)\}\}$$

Definition

$$X \rightarrowtail Y == \{f : X \pfun Y \mid \forall u, v : \mathrm{dom}\, f \bullet f(u) = f(v) \Rightarrow u = v\}$$
$$X \rightarrowtail Y == X \rightarrowtail Y \cap X \fun Y$$

Laws

$$(\alpha \rightarrowtail \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$
$$(\alpha \rightarrowtail \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \rightarrowtail \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \rightarrowtail \beta)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \rightarrowtail \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \rightarrowtail \beta)\downarrow$$
$$\forall x : \mathbb{P} X;\ y : \mathbb{P} Y;\ f : X \rightarrowtail Y \bullet f \in x \rightarrowtail y \Rightarrow f \in x \pfun y \land f \in x \rightarrowtail y \Rightarrow f^{\sim} \in y \pfun x$$
$$\forall x : \mathbb{P} X;\ y : \mathbb{P} Y;\ f : X \rightarrowtail Y \bullet f \in x \rightarrowtail y \Rightarrow f \in x \fun y \land f \in x \rightarrowtail y \Rightarrow f^{\sim} \in y \pfun x$$

5.4.3 Partial and Total Surjections

Name

$\_ \twoheadrightarrow \_$ $\_$ -|->> $\_$ – Partial surjections (onto)
$\_ \twoheadrightarrow \_$ $\_$ -->> $\_$ – Total surjections (onto)

Description
The set $\alpha \twoheadrightarrow \beta$ is the set of all surjective (onto) functions from the set α to the set β. The set $\alpha \twoheadrightarrow \beta$ is the set of all total surjective (onto) functions from the set α to the set β.

Examples

$$\begin{aligned}\{a, b, c\} \twoheadrightarrow \{d, e\} = \{&\{(a \mapsto d), (b \mapsto e)\}, \{(a \mapsto e), (b \mapsto d)\}, \ldots,\\ &\{(b \mapsto d), (c \mapsto e)\}, \{(b \mapsto e), (c \mapsto d)\},\\ &\{(a \mapsto d), (b \mapsto d), (c \mapsto e)\}, \ldots,\\ &\{(a \mapsto d), (b \mapsto e), (c \mapsto e)\}\}\\ \{a, b, c\} \twoheadrightarrow \{c, d\} = \{&\{(a \mapsto d), (b \mapsto d), (c \mapsto e)\}, \ldots,\\ &\{(a \mapsto d), (b \mapsto e), (c \mapsto e)\}\}\end{aligned}$$

Definition

$$X \twoheadrightarrow Y == \{f : X \pfun Y \mid \mathrm{ran} f = Y\}$$
$$X \twoheadrightarrow Y == X \twoheadrightarrow Y \cap X \fun Y$$

Laws

$$(\alpha \twoheadrightarrow \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$
$$(\alpha \twoheadrightarrow \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \twoheadrightarrow \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \twoheadrightarrow \beta)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \twoheadrightarrow \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \twoheadrightarrow \beta)\downarrow$$
$$\forall x : \mathbb{P} X;\ y : \mathbb{P} Y;\ f : X \twoheadrightarrow Y \bullet f \in x \twoheadrightarrow y \Rightarrow f \circ f^{\sim} \in \mathrm{id}\, y$$
$$\forall f : X \twoheadrightarrow Y \bullet \mathrm{dom} f \subseteq X \land \mathrm{ran} f = Y$$
$$\forall f : X \twoheadrightarrow Y \bullet \mathrm{dom} f = X \land \mathrm{ran} f = Y$$

5.4.4 Bijections

Name

$\_ \rightarrowtail \_$ `_ >->> _` – Bijections (one-to-one and onto)

Description
The set $\alpha \rightarrowtail \beta$ is the set of all bijective (one-to-one and onto) functions from the set α to the set β.

Examples

$$\{a, b\} \rightarrowtail \{c, d\} = \{\{(a \mapsto c), (b \mapsto d)\}, \{(a \mapsto d), (b \mapsto c)\}\}$$

Definition

$$X \rightarrowtail Y == X \twoheadrightarrow Y \cap X \rightarrowtail Y$$

Laws

$$(\alpha \rightarrowtail \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P}\, l \land \beta : \mathbb{P}\, r$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \rightarrowtail \beta)\downarrow$$
$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \rightarrowtail \beta)\downarrow$$
$$\forall x : \mathbb{P}\, X;\ y : \mathbb{P}\, Y;\ f : X \twoheadrightarrow Y \bullet$$
$$\quad f \in x \rightarrowtail y \Rightarrow f^{\sim} \in y \rightarrow x$$
$$\forall f : X \rightarrowtail Y \bullet$$
$$\quad \mathrm{dom} f = X \land$$
$$\quad \mathrm{ran} f = Y$$

5.4.5 Finite Functions

Name

$\_ \pfun \_$ `_ -||-> _` – Partial finite functions
$\_ \pinj \_$ `_ >-||-> _` – Finite injections (one-to-one)

Description
The set $\alpha \pfun \beta$ is the set of all finite functions from the set α to the set β. The set $\alpha \pinj \beta$ is the set of all finite injective (one-to-one) functions from the set α to the set β.

Examples

$$\{a, b\} \pfun \{c\} = \{\{\}, \{(a \mapsto c)\}, \{(b \mapsto c)\}, \{(a \mapsto c), (b \mapsto c)\}\}$$
$$\{a, b\} \pinj \{c, d\} = \{\{\}, \{(a \mapsto c)\}, \{(b \mapsto c)\}, \{(a \mapsto c), (b \mapsto d)\}, \{(a \mapsto d)\}, \{(b \mapsto d)\}, \{(a \mapsto d), (b \mapsto c)\}\}$$

Definition

$$X \pfun Y == X \fun Y \cap \mathbb{F}(X \times Y)$$
$$X \pinj Y == X \finj Y \cap X \pfun Y$$

Laws

$$(\alpha \pfun \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$
$$(\alpha \pinj \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \pfun \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \pfun \beta)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \pinj \beta)\downarrow$$
$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \pinj \beta)\downarrow$$
$$\forall x : \mathbb{P} X, y : \mathbb{P} Y \bullet x \pfun y = \{f : x \fun y \mid f \in \mathbb{F}(x \times y)\}$$
$$\forall x : \mathbb{P} X;\ y : \mathbb{P} Y;\ f : X \pinj Y \bullet f \in x \pinj y \Rightarrow f \in x \pfun y \land f \in x \pinj y \Rightarrow f^{\sim} \in y \pfun x$$

5.4.6 Function Overriding

Name

$\_ \oplus \_$ $\quad \_$`func_override`$\_$ – Function override

Description
The function $\alpha \oplus \beta$ is the merge of the functions α and β. Where the domains of the functions overlap β is given priority.

Examples

$$\{(a \mapsto b), (c \mapsto d)\} \oplus \{(a \mapsto e), (f \mapsto g)\} = \{(a \mapsto e), (c \mapsto d), (f \mapsto g)\}$$

Definition

$[X, Y]$

$$\_ \oplus \_ : (X \pfun Y) \times (X \pfun Y) \to (X \pfun Y)$$

$$\forall f, g : X \pfun Y \bullet f \oplus g = ((\operatorname{dom} g) \ndres f) \cup g$$

Laws

$$(\alpha \oplus \beta)\downarrow \Leftrightarrow \exists l, r \bullet \alpha : \mathbb{P} l \land \beta : \mathbb{P} r$$

$$\neg\, \alpha\downarrow \Rightarrow \neg\, (\alpha \oplus \beta)\downarrow$$

$$\neg\, \beta\downarrow \Rightarrow \neg\, (\alpha \oplus \beta)\downarrow$$

$$\forall f : X \pfun Y \bullet f \oplus f = f$$

$$\forall f : X \pfun Y \bullet f \oplus \varnothing = f \land \varnothing \oplus f = f$$

$$\forall f, g, h : X \pfun Y \bullet f \oplus (g \oplus h) = (f \oplus g) \oplus h$$

$$\forall f, g : X \pfun Y \bullet \operatorname{disjoint}\langle \operatorname{dom} f, \operatorname{dom} g\rangle \Rightarrow f \oplus g = f \cup g$$

$$\forall x : X;\ f, g : X \pfun Y \bullet x \in \operatorname{dom} f \setminus \operatorname{dom} g \Rightarrow (f \oplus g)(x) = f(x) \land x \in \operatorname{dom} g \Rightarrow (f \oplus g)(x) = g(x)$$

$$\forall f, g : X \pfun Y \bullet \operatorname{dom}(f \oplus g) = (\operatorname{dom} f) \cup (\operatorname{dom} g) \land \operatorname{ran}(f \oplus g) = (\operatorname{ran}((\operatorname{dom} g) \ndres f)) \cup (\operatorname{ran} g)$$

$$\forall x : \mathbb{P} X;\ f, g : X \pfun Y \bullet s \lhd (f \oplus g) = (s \lhd f) \oplus (s \lhd g) \land (f \oplus g) \rhd s = (f \rhd s) \oplus (g \rhd s)$$

5.5 Finite Sets

5.5.1 Finite Power Sets

Name

$\mathbb{F}\_$	`finite_`	– Finite power sets
$\mathbb{F}_1\_$	`finite_1_`	– Nonempty finite power sets

Description
The set $\mathbb{F}\,\alpha$ is the set of all finite subsets of the set α. The set $\mathbb{F}_1\,\alpha$ is the set of all finite nonempty subsets of the set α.

Examples

$\mathbb{F}\{a, b\} = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$
$\mathbb{N} \notin \mathbb{F}\,\mathbb{N}$
$\mathbb{F}_1\{a, b\} = \{\{a\}, \{b\}, \{a, b\}\}$
$\mathbb{N} \notin \mathbb{F}_1\,\mathbb{N}$

Definition

$$\mathbb{F}\,X == \{x : \mathbb{P}\,X \mid (\exists\, n : \mathbb{N}; f : (1\,..\,n) \rightarrowtail x \bullet \text{true})\}$$
$$\mathbb{F}_1\,X == \mathbb{F}\,X \setminus \{\varnothing\}$$

Laws

$$(\mathbb{F}\,\alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : \mathbb{P}\,t$$
$$(\mathbb{F}_1\,\alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : \mathbb{P}\,t$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ \mathbb{F}\,\alpha\downarrow$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ \mathbb{F}_1\,\alpha\downarrow$$
$$\forall\, x : \mathbb{P}\,X \bullet \varnothing \in \mathbb{F}\,x \land \varnothing \notin \mathbb{F}_1\,x$$

5.5.2 Cardinality

Name

$\#\_$ $\#\_$ – Set size

Description
The number $\#\alpha$ is the number of elements in the finite set α.

Examples

$$\#\{a, b\} = 2$$

provided that $a \neq b$.

Definition

$$[X]$$
$$\#\_ : \mathbb{F}\, X \to \mathbb{N}$$
$$\forall\, x : \mathbb{F}\, X \bullet$$
$$\quad \#x = \mu\, n : \mathbb{N} \mid (\exists f : 1 \mathinner{..} n \rightarrowtail x \bullet \text{true})$$

Laws

$$(\#\alpha)\downarrow \Leftrightarrow \exists\, x \bullet \alpha : \mathbb{P}\, x$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ \#\alpha\downarrow$$
$$\#\varnothing = 0$$
$$\forall\, x, y : \mathbb{F}\, X \bullet$$
$$\quad \#(x \cup y) = \#x + \#y - \#(x \cap y) \wedge$$
$$\quad \#(x \cap y) = \#x + \#y - \#(x \cup y) \wedge$$
$$\quad \#(x \times y) = \#x * \#y \wedge$$
$$\quad \#(x \setminus y) = \#x - \#(x \cap y)$$
$$\forall\, x : \mathbb{F}\, X \bullet$$
$$\quad \#(\mathbb{P}\, x) = 2^{\#x} \wedge$$
$$\quad \#(\mathbb{F}\, x) = 2^{\#x}$$

5.6 Sequences

5.6.1 Sequences

Name

$\mathrm{seq}\_$	`seq_`	– Finite sequences
$\mathrm{seq}_1\_$	`seq_1_`	– Nonempty finite sequences
$\mathrm{iseq}\_$	`iseq_`	– Injective (nonrepeating) sequences

Description
The set $\mathrm{seq}\,\alpha$ is the set of all finite sequences of members of the set α (i.e. the set of all total functions from some set $1 \ldots n$ of natural numbers to α). The set $\mathrm{seq}_1\,\alpha$ is the set of all nonempty finite sequences of members of the set α. The set $\mathrm{iseq}\,\alpha$ is the set of all injective (nonrepeating) finite sequences of members of the set α.

Examples

$$\begin{aligned}
\mathrm{seq}\{a,b\} &= \{\{\}, \{1 \mapsto a\}, \{1 \mapsto b\}, \{1 \mapsto a, 2 \mapsto a\}, \{1 \mapsto a, 2 \mapsto b\},\\
&\qquad \{1 \mapsto b, 2 \mapsto a\}, \{1 \mapsto b, 2 \mapsto b\}, \ldots\}\\
\mathrm{seq}_1\{a,b\} &= \{\{1 \mapsto a\}, \{1 \mapsto b\}, \{1 \mapsto a, 2 \mapsto a\}, \{1 \mapsto a, 2 \mapsto b\},\\
&\qquad \{1 \mapsto b, 2 \mapsto a\}, \{1 \mapsto b, 2 \mapsto b\}, \ldots\}\\
\mathrm{iseq}\{a,b\} &= \{\{\}, \{1 \mapsto a\}, \{1 \mapsto b\}, \{1 \mapsto a, 2 \mapsto b\}, \{1 \mapsto b, 2 \mapsto a\}\}
\end{aligned}$$

Definition

$$\begin{aligned}
&\mathrm{seq}\,X == \{s : \mathbb{N} \pfun X \bullet \mathrm{dom}\,s = 1 \ldots \#s\}\\
&\mathrm{seq}_1\,X == \mathrm{seq}\,X \setminus \{\varnothing\}\\
&\mathrm{iseq}\,X == \mathrm{seq}\,X \cap (\mathbb{N} \pinj X)
\end{aligned}$$

Laws

$$\begin{aligned}
&(\mathrm{seq}\,\alpha)\downarrow \Leftrightarrow \exists\, x \bullet \alpha : \mathbb{P}\,x\\
&(\mathrm{seq}_1\,\alpha)\downarrow \Leftrightarrow \exists\, x \bullet \alpha : \mathbb{P}\,x\\
&(\mathrm{iseq}\,\alpha)\downarrow \Leftrightarrow \exists\, x \bullet \alpha : \mathbb{P}\,x
\end{aligned}$$

$$\begin{aligned}
&\neg\ \alpha\downarrow \Rightarrow \neg\ \mathrm{seq}\,\alpha\downarrow\\
&\neg\ \alpha\downarrow \Rightarrow \neg\ \mathrm{seq}_1\,\alpha\downarrow\\
&\neg\ \alpha\downarrow \Rightarrow \neg\ \mathrm{iseq}\,\alpha\downarrow
\end{aligned}$$

$$\begin{aligned}
&\mathrm{seq}_1\,X \subseteq \mathrm{seq}\,X\\
&\mathrm{iseq}\,X \subseteq \mathrm{seq}\,X
\end{aligned}$$

$$\begin{aligned}
&\forall s : \mathrm{iseq}\,X;\ n, m : \mathrm{dom}\,s \bullet\\
&\qquad n \neq m \Leftrightarrow s(n) \neq s(m)
\end{aligned}$$

5.6.2 Concatenation

Name

$\_ \frown \_$	`_concat_`	– Concatenation
$\frown / \_$	`d_concat_`	– Generalized (distributed) concatenation

Description

The sequence $\alpha \frown \beta$ is the sequence α followed by (concatenated to) the sequence β. The sequence $\frown / \alpha$ is the sequence made by concatenating all the sequences in the sequence α.

Examples

$\langle a, b \rangle \frown \langle c, d \rangle = \langle a, b, c, d \rangle$
$\langle \langle a, b \rangle, \langle c, d \rangle, \langle e, f \rangle \rangle = \langle a, b, c, d, e, f \rangle$

Definition

$[X]$

$$\_ \frown \_ : \operatorname{seq} X \times \operatorname{seq} X \pfun \operatorname{seq} X$$
$$\frown / \_ : \operatorname{seq} \operatorname{seq} X \pfun \operatorname{seq} X$$

$$\forall s, t : \operatorname{seq} X \bullet$$
$$s \frown t = s \cup \{ n : \mathbb{N};\ e : \operatorname{ran} t \mid (\exists i : \operatorname{dom} t \bullet n = i + \# s \land e = t(i)) \}$$
$$\frown / \langle \rangle = \langle \rangle$$
$$\forall s : \operatorname{seq} X;\ t : \operatorname{seq} \operatorname{seq} X \bullet$$
$$\frown / (\langle s \rangle \frown t) = s \frown (\frown / t)$$

Laws

$$(\alpha \frown \beta) \downarrow \Leftrightarrow \exists t \bullet (\alpha : \operatorname{seq} t \land \beta : \operatorname{seq} t)$$
$$(\frown / \alpha) \downarrow \Leftrightarrow \exists t \bullet \alpha : \operatorname{seq} \operatorname{seq} t$$
$$\neg\ \alpha \downarrow \Rightarrow \neg\ (\alpha \frown \beta) \downarrow$$
$$\neg\ \beta \downarrow \Rightarrow \neg\ (\alpha \frown \beta) \downarrow$$
$$\neg\ \alpha \downarrow \Rightarrow \neg\ (\frown / \alpha) \downarrow$$
$$\forall s : \operatorname{seq} X \bullet$$
$$\langle \rangle \frown s = s \land$$
$$s \frown \langle \rangle = s$$
$$\forall s, t, u : \operatorname{seq} X \bullet$$
$$s \frown (t \frown u) = (s \frown t) \frown u$$
$$\forall s, t : \operatorname{seq} X \bullet$$
$$\# (s \frown t) = \# s + \# t$$
$$\forall s : \operatorname{seq} X \bullet$$
$$\frown / \langle s \rangle = s$$
$$\forall s, t : \operatorname{seq} X \bullet$$
$$\frown / \langle s, t \rangle = s \frown t$$
$$\forall s, t : \operatorname{seq}_1 X \bullet$$
$$s \frown t \in \operatorname{seq}_1 X$$
$$\forall s : \operatorname{seq} X;\ t : \operatorname{seq}_1 X \bullet$$
$$s \frown t \in \operatorname{seq}_1 X$$
$$\forall s : \operatorname{seq}_1 X;\ t : \operatorname{seq} X \bullet$$
$$s \frown t \in \operatorname{seq}_1 X$$

5.6.3 Destructors

Name

head _	`head_`	– First element of a sequence
tail _	`tail_`	– All but the first element of a sequence
last _	`last_`	– Last element of a sequence
front _	`front_`	– All but the last element of a sequence

Description
The element *head* α is the first element in the nonempty sequence α. The sequence *tail* α is composed of all but the first element of the nonempty sequence α. The element *last* α is the last element in the nonempty sequence α. The sequence *front* α is composed of all but the last element of the nonempty sequence α.

Examples

$$head\langle a, b, c\rangle = a$$
$$tail\langle a, b, c\rangle = \langle b, c\rangle$$
$$last\langle a, b, c\rangle = c$$
$$front\langle a, b, c\rangle = \langle a, b\rangle$$

Definition

$[X]$

$$head\_ : \mathrm{seq}_1 X \pfun X$$
$$tail\_ : \mathrm{seq}_1 X \fun \mathrm{seq}\, X$$
$$last\_ : \mathrm{seq}_1 X \pfun X$$
$$front\_ : \mathrm{seq}_1 X \fun \mathrm{seq}\, X$$

$$\forall s : \mathrm{seq}_1 X \bullet head\, s = s(1)$$

$$\forall s : \mathrm{seq}_1 X \bullet tail\, s = \mu\, t : seqX \mid s = \langle head\, s\rangle \frown t$$

$$\forall s : \mathrm{seq}_1 X \bullet last\, s = s(\#s)$$

$$\forall s : \mathrm{seq}_1 X \bullet front\, s = \mu\, t : seqX \mid s = t \frown \langle last\, s\rangle$$

Laws

$$(head\, \alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : \mathrm{seq}_1 t$$
$$(tail\, \alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : \mathrm{seq}_1 t$$
$$(last\, \alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : \mathrm{seq}_1 t$$
$$(front\, \alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : \mathrm{seq}_1 t$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (head\, \alpha)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (tail\, \alpha)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (last\, \alpha)\downarrow$$
$$\neg\, \alpha\downarrow \Rightarrow \neg\, (front\, \alpha)\downarrow$$
$$\neg\, head\langle\,\rangle\downarrow$$
$$\neg\, tail\langle\,\rangle\downarrow$$
$$\neg\, last\langle\,\rangle\downarrow$$
$$\neg\, front\langle\,\rangle\downarrow$$

$$
\begin{array}{l}
\forall\, s : \mathrm{seq}_1\, X \bullet \\
\quad \langle head\, s \rangle \frown (tail\, s) = s \land \\
\quad (front\, s) \frown \langle last\, s \rangle = s \\
\forall\, x : X;\ s : \mathrm{seq}\, X \bullet \\
\quad head(\langle x \rangle \frown s) = x \land \\
\quad tail(\langle x \rangle \frown s) = s \land \\
\quad last(s \frown \langle x \rangle) = x \land \\
\quad front(s \frown \langle x \rangle) = s
\end{array}
$$

5.6.4 Reverse

Name

*rev*_ `rev_` – Reverse of a sequence

Description
The sequence $rev\,\alpha$ is the sequence α reversed.

Examples

$$rev\langle a, b, c\rangle = \langle c, b, a\rangle$$

Definition

$$[X]$$

$$rev\_ : \operatorname{seq} X \rightarrowtail \operatorname{seq} X$$

$$rev\langle\,\rangle = \langle\,\rangle$$
$$\forall\, x : X;\ s : \operatorname{seq} X \bullet$$
$$\quad rev(\langle x\rangle \frown s) = (rev\, s) \frown \langle x\rangle$$

Laws

$$(rev\,\alpha)\downarrow \Leftrightarrow \exists\, t \bullet \alpha : \operatorname{seq} t$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ (rev\,\alpha)\downarrow$$
$$\forall\, x : X \bullet$$
$$\quad rev\langle x\rangle = \langle x\rangle$$
$$\forall\, s : \operatorname{seq} X \bullet$$
$$\quad rev\, rev\, s = s$$
$$\forall\, s, t : \operatorname{seq} X \bullet$$
$$\quad rev(s \frown t) = (rev\, t) \frown (rev\, s)$$
$$\forall\, s : \operatorname{seq}_1 X \bullet$$
$$\quad head(rev\, s) = last\, s \wedge$$
$$\quad tail(rev\, s) = rev\, front\, s \wedge$$
$$\quad last(rev\, s) = head\, s \wedge$$
$$\quad front(rev\, s) = rev\, tail\, s$$
$$\forall\, s : \operatorname{seq} X;\ f : \mathbb{P}\, X \bullet$$
$$\quad (rev\, s) \upharpoonright f = rev(s \upharpoonright f)$$

5.6.5 Filtering

Name

$\_ \upharpoonright \_$ `_filter_` – Filtered sequence

Description
The sequence $\alpha \upharpoonright \beta$ is the sequence α where elements not in the set β have been removed.

Examples

$$\langle a, b, c, b \rangle \upharpoonright \{a, b\} = \langle a, b, b \rangle$$

Definition

$$[X]$$

$$\_ \upharpoonright \_ : \operatorname{seq} X \times \mathbb{P} X \twoheadrightarrow \operatorname{seq} X$$

$$\begin{array}{l}
\forall f : \mathbb{P} X \bullet \\
\quad \langle\,\rangle \upharpoonright f = \langle\,\rangle \land \\
\quad \forall x : X;\ s : \operatorname{seq} X \bullet \\
\qquad x \in f \Rightarrow (\langle x \rangle \frown s) \upharpoonright f = \langle x \rangle \frown (s \upharpoonright f) \land \\
\qquad x \notin f \Rightarrow (\langle x \rangle \frown s) \upharpoonright f = (s \upharpoonright f)
\end{array}$$

Laws

$$(\alpha \upharpoonright \beta)\downarrow \Leftrightarrow \exists t \bullet (\alpha : \operatorname{seq} t \land \beta : \mathbb{P} t)$$

$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \upharpoonright \beta)\downarrow$$

$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \upharpoonright \beta)\downarrow$$

$$s \upharpoonright \varnothing = \langle\,\rangle$$

$$\begin{array}{l}
\forall s : \operatorname{seq} X;\ f : \mathbb{P} X \bullet \\
\quad (s \upharpoonright f) \upharpoonright f = s \upharpoonright f
\end{array}$$

$$\begin{array}{l}
\forall s : \operatorname{seq} X;\ f, g : \mathbb{P} X \bullet \\
\quad (s \upharpoonright f) \upharpoonright g = (s \upharpoonright g) \upharpoonright f \land \\
\quad (s \upharpoonright f) \upharpoonright g = s \upharpoonright (f \cap g)
\end{array}$$

$$\begin{array}{l}
\forall s : \operatorname{seq} X;\ f : \mathbb{P} X \bullet \\
\quad \operatorname{ran} s \subseteq f \Leftrightarrow s \upharpoonright f = s
\end{array}$$

5.6.6 Disjointness and Partitions

Name

disjoint_	`disjoint_`	– Is disjoint
partitions	`_partitions_`	– Is a partition of

Description

The predicate disjoint α means each pair of elements in ran α share no elements. The predicate α partitions β means that disjoint α and $\bigcup \operatorname{ran} \alpha = \beta$. Normally, α will be a sequence.

Examples

$\operatorname{disjoint}\langle \{a, b\}, \{c, d\} \rangle$
$\neg$ $\operatorname{disjoint}\langle \{a, b\}, \{a, c, d\} \rangle$
$\langle \{a, b\}, \{c, d\} \rangle$ partitions $\{a, b, c, d\}$
$\neg$ $(\langle \{a, b\}, \{c, d\} \rangle$ partitions $\{a, b, c, d, e\})$
$\neg$ $(\langle \{a, b\}, \{c, d\} \rangle$ partitions $\{a, b, c, \})$

Definition

$[X]$

$\operatorname{disjoint}\_ : \mathbb{P}(\operatorname{seq} \mathbb{P} X)$
$\_\text{partitions}\_ : (\operatorname{seq} \mathbb{P} X) \leftrightarrow \mathbb{P} X$

$\forall f : I \pfun \mathbb{P} X \bullet$
$\quad \operatorname{disjoint} f \Leftrightarrow \forall i, j : \operatorname{dom} f \bullet i \neq j \Rightarrow f(i) \cap f(j) = \varnothing$
$\forall f : I \pfun \mathbb{P} X;\ t : \mathbb{P} X \bullet$
$\quad f$ partitions $t \Leftrightarrow \operatorname{disjoint} f \land \bigcup \operatorname{ran} f = t$

Laws

$(\operatorname{disjoint} \alpha)\downarrow \Leftrightarrow \exists x, y \bullet \alpha : x \pfun \mathbb{P} y$
$(\alpha$ partitions $\beta)\downarrow \Leftrightarrow \exists x, y \bullet (\alpha : x \pfun \mathbb{P} y \land \beta : \mathbb{P} y)$
$\neg\ \alpha\downarrow \Rightarrow \neg\ (\operatorname{disjoint} \alpha)\downarrow$
$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha$ partitions $\beta)\downarrow$
$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha$ partitions $\beta)\downarrow$
$\operatorname{disjoint} \varnothing$
$\forall x : \mathbb{P} X \bullet$
$\quad \operatorname{disjoint}\langle x \rangle$
$\forall x, y : \mathbb{P} X \bullet$
$\quad x \cap y = \varnothing \Leftrightarrow \operatorname{disjoint}\langle x, y \rangle$
$\forall x, y : \operatorname{seq} X \bullet$
$\quad \operatorname{disjoint}(x \frown y) \Rightarrow \operatorname{disjoint} x \land \operatorname{disjoint} y$
$\forall x, y, z : \mathbb{P} X \bullet$
$\quad x \cap y = \varnothing \land x \cup y = z \Leftrightarrow \langle x \rangle$ partitions z

5.7 Bags

5.7.1 Bags

Name

$\mathrm{bag}\_$ $\quad$ `bag_` $\quad$ – Set of bags

Description
The set $\mathrm{bag}\,\alpha$ is the set of all bags containing members of the set α.

Examples

$[\![a, a, b]\!] \in \mathrm{bag}\{a, b\}$
$[\![a, a, b]\!] = \{a \mapsto 2, b \mapsto 1\}$

Definition

$\mathrm{bag}\, X == X \pfun \mathbb{N}_1$

Laws

$(\mathrm{bag}\,\alpha)\downarrow \Leftrightarrow \exists\, x \bullet \alpha : x$
$\neg\ \alpha\downarrow \Rightarrow \neg\ \mathrm{bag}\,\alpha\downarrow$

5.7.2 Membership

Name

count _ _	`count(_,_)`	– Number of members
_in_bag_	`_in_bag_`	– Bag membership

Description
The number *count* α β is the number times β appears in the bag α. The predicate α in_bag β means the entity α is in the bag β.

Examples

$$count\ [\![a, a, b]\!]\ a = 2$$
$$a\ \text{in\_bag}\ [\![a, a, b]\!]$$

Definition

$$[X]$$
$$count : \text{bag}\, X \times X \to \mathbb{N}$$
$$\_\text{in\_bag}\_ : X \leftrightarrow \text{bag}\, X$$
$$\forall x : X;\ b : \text{bag}\, X \bullet$$
$$x \,\text{in\_bag}\, b \Leftrightarrow x \in \text{dom}\, b$$
$$\forall b : \text{bag}\, X;\ x : X \bullet$$
$$count\, b\, x = \textbf{if}\, x \in b\, \textbf{then}\, b(x)\, \textbf{else}\, 0$$

Laws

$$(count\ \alpha)\downarrow \Leftrightarrow \exists x \bullet \alpha : \text{bag}\, x$$
$$(\alpha\ \text{in\_bag}\ \beta)\downarrow \Leftrightarrow \exists x \bullet (\alpha : x \land \beta : \text{bag}\, x)$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ count\ \alpha\downarrow$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ \alpha\ \text{in\_bag}\ \beta\downarrow$$
$$\neg\ \beta\downarrow \Rightarrow \neg\ \alpha\ \text{in\_bag}\ \beta\downarrow$$
$$\forall b : \text{bag}\, X;\ x : X \bullet$$
$$x \,\text{in\_bag}\, b \Leftrightarrow count\, b\, x > 0$$
$$\forall x : X \bullet$$
$$count [\![\,]\!]\, x = 0 \land$$
$$count [\![x]\!]\, x = 1$$
$$\forall b, c : \text{bag}\, X;\ x : X \bullet$$
$$count (b \uplus c)\, x = count\, b\, x + count\, c\, x$$
$$\forall b, c : \text{bag}\, X;\ x : X \bullet$$
$$x \,\text{in\_bag}\, b \Rightarrow x \,\text{in\_bag} (b \uplus c)$$

5.7.3 Bag Union

Name

$\_ \uplus \_$ $\quad$ $\_ \texttt{bag\_union} \_$ $\quad$ – Bag union (merge)

Description
The bag $\alpha \uplus \beta$ is the merge of the bags α and β.

Examples

$$[\![a, a, b]\!] \uplus [\![b, c]\!] = [\![a, a, b, b, c]\!]$$

Definition

$$[X]$$

$$\_ \uplus \_ : \operatorname{bag} X \times \operatorname{bag} X \pfun \operatorname{bag} X$$

$$\begin{array}{l} \forall\, b, c : \operatorname{bag} X \bullet \\ \quad b \uplus c = \{x : \operatorname{dom} b \cup \operatorname{dom} c;\ y : \mathbb{N}_1 \mid \\ \qquad\qquad y = \mathit{count}\, b\, x + \mathit{count}\, c\, x\} \end{array}$$

Laws

$$(\alpha \uplus \beta)\downarrow \Leftrightarrow \exists\, t \bullet \alpha, \beta : \operatorname{bag} t$$

$$\neg\ \alpha\downarrow \Rightarrow \neg\ (\alpha \uplus \beta)\downarrow$$

$$\neg\ \beta\downarrow \Rightarrow \neg\ (\alpha \uplus \beta)\downarrow$$

$$\begin{array}{l} \forall\, b : \operatorname{bag} X \bullet \\ \quad [\![]\!] \uplus b = b \land \\ \quad b \uplus [\![]\!] = b \end{array}$$

$$\begin{array}{l} \forall\, b, c : \operatorname{bag} X \bullet \\ \quad b \uplus c = c \uplus b \end{array}$$

$$\begin{array}{l} \forall\, b, c, d : \operatorname{bag} X \bullet \\ \quad (b \uplus c) \uplus d = c \uplus (b \uplus d) \end{array}$$

$$\begin{array}{l} \forall\, b, c : \operatorname{bag} X \bullet \\ \quad \operatorname{dom}(b \uplus c) = \operatorname{dom} b \cup \operatorname{dom} c \end{array}$$

5.7.4 Sequence to Bag Conversion

Name

items _ `items` _ – Sequence to bag conversion

Description
The bag *items* α is the bag corresponding to the sequence α (i.e. each member of *items* α appears the same number of times in α and vice versa).

Examples

$items\langle a, a, b\rangle = [\![a, a, b]\!]$

Definition

$[X]$

$$items\_ : \operatorname{seq} X \to \operatorname{bag} X$$

$$\forall s : \operatorname{seq} X \bullet items\, s = \{x : \operatorname{ran} s;\ y : \mathbb{N}_1 \mid y = \#(s \rhd \{x\})\}$$

Laws

$$(items\, \alpha)\downarrow \Leftrightarrow \exists t \bullet \alpha : \operatorname{seq} t$$
$$\neg\ \alpha\downarrow \Rightarrow \neg\ items\, \alpha\downarrow$$
$$items\langle\,\rangle = [\![\,]\!]$$
$$\forall s, t : \operatorname{seq} X \bullet items(s \frown t) = (items\, s) \uplus (items\, t)$$

5.8 Computational types of the intermediate language

5.8.1 Integers and Subsets of Integers

Names

`integer`
`natural`
`natural_1`

Description The computational types `integer`, `natural`, and `natural_1` are defined as subtypes of the theoretical (infinite) `int` type and are represented by a specific (machine dependent) range of integers.

Definitions

integer_first, *integer_last* : `int` ;
`integer` $==$ {`dec` i : `int` $\mid$ *integer_first* $<=$ i `and` $i <=$ *integer_last*};
`natural` $==$ {`dec` i : `integer` $\mid i >= 0$};
`natural_1` $==$ {`dec` i : `integer` $\mid i > 0$};

5.8.2 Computational integer operators

Names

subc	– Integer subtraction
addc	– Integer addition
multc	– Integer multiplication
divc	– Integer division
modc	– Integer remainder on division
expc	– Integer exponentiation
ltc	– Integer less than
ltec	– Integer less than or equal
gtc	– Integer greater than
gtec	– Integer greater than or equal
uptoc	– Consecutive integer constructor
eqc	– Integer equals
neqc	– Integer not equals

Description Machine oriented arithmetic operators are defined so that results are only guaranteed to be defined when results are within range.

Definition These operations are partial functions on $\mathbb{Z}$, the theoretical integer type. For example, `addc` is a computational operator specified as follows.

$$addc : \mathbb{Z} \times \mathbb{Z} \pfun \mathbb{Z}$$

$$\forall x, y : \mathbb{Z} \bullet (integer\_first \leq x + y \land x + y \leq integer\_last) \Rightarrow x \ \texttt{addc}\ y = x + y$$

5.8.3 Computational boolean operators

Names

eqc	– Boolean equals
neqc	– Boolean not equals
andc	– Boolean and
orc	– Boolean or
xorc	– Boolean exclusive or

Description The computational boolean type is modelled by the logical boolean type, but with different operators. So, for example, when using the logical operators, $p \vee \neg\, p$ is always true, using computational operators, this may not be so, since the subexpression p may diverge in some way. These computational boolean operators can be regarded as partial operations on the boolean type.

The generic computational equality and inequality operators are strict, that is, produce boolean values only if both operands are well defined (produce normal values).

Definition

```
axiom is
dec
    andc : bool − − > bool
    orc : bool − − > bool
    xorc : bool − − > bool
pred
    forall x, y : bool @
            x andc y = x and y;
    forall x, y : bool @
            x orc y = x orc y;
    forall x, y : bool @
            x xorc y = x xorc y
end


axiom [T] is
dec
    eqc, neqc : T cross T
pred
    forall x, y : T @ ( x eqc y ) ⇔ ( x = y );
    forall x, y : T @ ( x neqc y ) ⇔ ( x ≠ y )
end
```

5.8.4 Constrained array types

Name

$\texttt{array}\,(I, R)$ – an array with index type I and range R

Description Constrained array types are represented in the intermediate language by total functions from finite, discrete types to intermediate language types. Further description is to be found in B.4.3.

Definition

$[D, R]$

$$\texttt{array} : (\mathbb{P}\, D) \times (\mathbb{P}\, R) \rightarrow \mathbb{P}(D \rightarrow R)$$

Appendix A. Concrete ASCII Syntax

A.1 Introduction

This appendix gives the concrete ASCII syntax for Sum.

Comments are not included in the grammar, but may be used anywhere within a specification document. A double slash indicates that the remainder of the line is a comment:

```
// comment goes here
```

The notation used is as follows:

Nonterminal Nonterminals appear in *slanted italic font*.

`Terminal` Terminals appear in `underlined typewriter font`. Observe that underscore is itself as a terminal symbol; it looks like this: `_`.

$[\alpha]$ The subproduction α is optional and thus may be elided.

$\{\alpha\}$ The subproduction α may be repeated zero or more times.

$\alpha \to \beta$ The nonterminal α may be replaced by β.

$\alpha \mid \beta$ Either the subproduction α or β may be used.

A.2 Specification

Specification $\to$ *ModuleItemList*

A.3 Modules

ModuleItemList $\to$ *ModuleItem* {*ModuleItem*}

ModuleItem $\to$ `module` *Id* [`(` *FormalList* `)`] [[*PredList*]] `is` [*DeclList*]`end` *Id*

FormalList $\to$ *Formal* {`;` *Formal* }

Formal $\to$ *Id* | *VarDecl*

A.4 Declarations

DeclList $\to$ *Declaration* {`;` *Declaration* }

Declaration → `import` *Id* [[`(` *ExpList* `)`] [`{` *Renames* `}`] `as` *Id*]
– module import

| [`op`]`schema` *Id* [[*SchFormalList*]] `is` [`dec` *BasicDeclList*]
[`pred` *PredList*] `end` *Id*
– schema definition

| `axiom` [[*SchFormalList*]] `is` [`dec` *VarDeclList*]
[`pred` *PredList*] `end`
– axiom declaration

| *ModuleItem*
– nested module

| [*DeclNameList*]
– given set

| *Id* == *Expression*
– abbreviation

| *Id* ::= *Branch* { | *Branch* }
– free type

| *Id* ::= `enum` (*Id* { , *Id* })
– enumerated type

| `visible` *Id* [{ *Selections* }]
– visible declaration

| `function` [*VarDeclList*] [`pred` *PredList*] `end`
– function definition in IL

| (−+ `ergo` `axiom` *Id* +−)
– override Ergo default name prefix

| (−− : `refinement` : *ErgoCommands* −−)
– include refinement commands

| (−− : `safety` : *ErgoCommands* −−)
– include safety commands

| `state_machine` == `module` (*Id* , *Id* , [*DeclNameList*])
– Ergo definition of the state machine

| *VarDecl*
| *Predicate*

Selections → *DeclName* { , *DeclName* }

VarDeclList → *VarDecl* { ; *VarDecl*}

VarDecl → *DeclNameList* : *Expression*

DeclNameList → *DeclName* { , *DeclName*}

SchFormalList → *Id* { , *Id* }

BasicDeclList → *BasicDecl* { ; *BasicDecl* }

BasicDecl → *Name*
| *Name* [*ExpList*]
| *VarDecl*

Branch → *Id* | << *Expression* >>

A.5 Predicates

PredList → *Predicate* { ; *Predicate* }

Predicate → *Quantifier SchemaText* @ *Predicate*
| *LogPredEquivalence*

Quantifier → forall | exists | exists$_1$
| var | sm$_1$ | let

LogPredEquivalence → *LogPredEquivalence* <=> *LogPredImplication*
| *LogPredImplication*

LogPredImplication → *LogPredOr* => *LogPredImplication*
| *LogPredOr*

LogPredOr → *LogPredOr* exor *LogPredAnd*
| *LogPredOr* or *LogPredAnd*
| *LogPredAnd*

LogPredAnd → *LogPredAnd* and *LogPredSeq*
| *LogPredSeq*

LogPredSeq → *LogPredSeq* ; ; *BasicPredicate*
| *BasicPredicate*

BasicPredicate → true
| false
| not *BasicPredicate*
| pre *BasicPredicate*
| changes_only { [*SchVarList*] }
| *Expression InRelOp_Ex Expression*
| *PreRelOp Expression*
| *CmpndSch*
| assign *SchVarList* := *ExpList*
| ifc *Predicate* then *Predicate* [else *Predicate*]fi
| while *Predicate* do *Predicate* done
| call (*Name* , ([*InputBindList*]) , ([*OutputBindList*]))

InRelOp_Ex → *InRelOp* | < | >
| ltc | ltec | gtc | gtec

InRelOp → = | /= | >= | <= | subset | p_subset | in | in_bag | not_in
| partitions | eqc | neqc

InputBindList → *Id* => *Expression* { , *Id* => *Expression* }

OutputBindList → *Id* => *Name* { , *Id* => *Name* }

SchVarList → *Name* { , *Name* }

PreRelOp → disjoint

AndPredList → *AndPredList* and *BasicPredicate*
| *BasicPredicate*

A.6 Schema Expressions / Predicates

CmpndSch → *CmpndSch* `s_compose` *CmpndSch*1 – composition
| *CmpndSch*1

*CmpndSch*1 → *CmpndSch*1 `BackSlash` (*SchVarList*) – hiding
| *CmpndSch*1 { *Renames* } – renaming
| *CmpndSch*2

*CmpndSch*2 → *CmpndSch*2 | ^ *BasicSch* – projection
| *BasicSch*

BasicSch → [*SchemaText*] – unnamed schema
| *Expression*

SchemaText → *BasicDeclList* [| *Predicate*]

A.7 Expressions

ExpList → *Expression* {, *Expression*}

Expression → `if` *Predicate* `then` *Expression* `else` *Expression* `fi`
| *Expression*0

*Expression*0 → *CartExp InGen Expression*0
| *CartExp*

CartExp → *Expression*1 `cross` *CartExp*
| *Expression*1

*Expression*1 → *Expression*1 *Infix Expression*2
| *Expression*2

Infix → | −− > | .. | + | * | `diff` | ^ | `div` | `mod` | `func_override` | `b_compose` | `f_compose`
| `dom_restrict` | `ran_restrict` | `dom_subtract` | `ran_subtract` | `union` | `bag_union` | `inter`
| `sym_diff` | `filter` | ** | −
| `subc` | `addc` | `multc` | `divc` | `modc` | `expc` | `uptoc` | `andc` | `orc` | `xorc`

*Expression*2 → *Prefix_Id Expression*2
| *BasicExp*

Prefix_Id → *Prefix* | _ | #

Prefix → `power`
| `power_1`
| `finite`
| `finite_1`
| `seq`
| `rev`
| `seq_1`
| `iseq`
| `bag`
| `id`
| `dom`
| `ran`
| `gen_union`
| `gen_inter`

BasicExp → *Name* – variable
| *Name* [*ExpList*] – schema instantiation
| *String*
| *Char*
| *Number*
| *Name* [*ExpList*] . *Id* – generic binding selection
| *BasicExp* (*ExpList*) – function application
| (*Expression* , *ExpList*) – tuple
| { [*ExpList*]} – set display
| { [`dec`]*SchemaText* [`@` *Expression*] } – set comprehension
| ≤ [*ExpList*]≥ – sequence display
| | [[*ExpList*]] | – bag display
| (`lambda` *SchemaText* `@` *Expression*) – lambda expression
| (`mu` *SchemaText* [`@` *Expression*]) – mu expression
| `theta` *Name* – theta expression
| `upd` (*Name* , *Expression* , *Expression*) – update array variable
| (*Expression* `is` *Expression* {, *Expression* `is` *Expression*}) – named array aggregate
| `the` *SchemaText* `@` (*AndPredList*) – record display
| (*Predicate*)

A.8 Names

Name → *Id*
| *Id* . *Name*

Opname → _ *InfixOp* _
| *PrefixOp* _

DeclName → *Id*
| *Opname*

InfixOp → *Infix*
| *InRelOp_Ex*

PrefixOp → *Prefix_Id*
| *PreRelOp*

Word → *Letter* {*Letter* | *Digit* | _ *Letter* | _ *Digit*}

Id → *Word* {! | ? | '}

Number → ... | `-1` | `0` | `1` | ...

Letter → `a` | `b` | ... | `z` | `A` | `B` | ... | `Z`

Digit → `0` | `1` | ... | `9`

Renames → *Rename* {, *Rename* }

Rename → *DeclName* / *Name*

Symbol → ! | @ | # | $ | ...
Character → *Letter* | *Digit* | *Symbol*
Char → '*Character*'
String → "[*Str*]"
Str → *Character*{*Character*}

A.9 Operator Precedence and Associativity

The following table gives both the relative precedence and associativity of Sum's operators. Thus, for example, `A and B or C` is implicitly bracketed as `(A and B) or C` since `and` has higher precedence than `or`.

Operators	Kind	Associativity
`<=>`	infix	left
`=>`	infix	right
`or`, `xor`,`orc`, `xorc`	infix	left
`and`, `andc`	infix	left
`not`	prefix	–
`=`, `/=`, `in`, `not_in`, `in_bag`, `<`, `>`, `>=`, `<=`, `subset`, `p_subset`, `eqc`, `neqc`, `ltc`, `ltec`, `gtc`, `gtec`	infix	left
`|-->`	infix	left
`..`, `uptoc`	infix	–
`+`, `-`, `diff`, `union`, `^`, `bag_union`, `subc`, `addc`	infix	left
`*`, `div`, `mod`, `inter`, `multc`, `divc`, `modc`	infix	left
`**`, `expc`	infix	right
`func_override`, `f_compose`, `b_compose`	infix	left
`dom_restrict`, `ran_restrict`, `dom_subtract`, `ran_subtract`	infix	left
`<-->`, `-|->`, `-->`, `>-|->`, `>-->`, `-|->>`, `-->>`, `>-->>`, `-||->`, `>-||->`	infix	right
`cross`	distfix	–
`-` (unary), #, `power`, `finite`, `power_1`, `finite_1`, `seq`, `seq_1`, `iseq`, `bag`, `dom`, `ran`, `gen_union`, `gen_inter`	prefix	–
Function Arg	prefix	left
Binding`.`*Field*, *Tuple*`.`*Number*	infix	left

The lowest precedence operators appear at the top of the table and the highest precedence operators at the bottom (i.e. looser binding above tighter binding).

Appendix B. Modules in The Cogito Methodology

B.1 Introduction

Modularity is crucial to the management of the software development process. It is the most important mechanism for managing complexity, achieving a separation of concerns and isolating changes. However, the features for achieving modularity must be chosen carefully since they have wide-ranging repercussions. As well as providing a means for structuring specifications, modularity also has implications for refinement and for structuring the associated theories for reasoning. In the Cogito Methodology, the key unit for maintenance of both specifications and proofs is the specification module.

Sum modules are managed by associating modules with units in a library. The importation information in a specification contributes to configuration information at the library level. The library, and its associated dependency information, is managed by the Cogito repository manager[21]. The repository manager maintains various types of dependencies between specifications, theories and proofs as well as other artifacts of the software development process.

The design of Sum modules has been partly influenced by the fact that specifications may be developed to Ada [22] code. The transition from detailed designs to implementations in Ada is made as straightforward as possible, with the modular structure of the detailed designs suggesting a package structure for the Ada implementations. Without these features, the projection to Ada implementations would require complex analysis and translation rules in order to ensure correct Ada is produced. The module mechanism is also minimal, in that extensions may be made to accommodate languages whose module reference semantics are more liberal than those used here. If a more liberal approach were taken, languages with a stricter notion of module reference would be very difficult to accommodate. The modular structure of the final program is not necessarily defined by the modular structure of the initial specification [11]. Changes to the modular structure of specifications may be made in the process of refinement to detailed designs and implementations.

Modules in Sum perform two roles. First, they provide an encapsulation mechanism which allows a state and its associated operations to be defined (conceptually corresponding to a state machine). Second, they define the theory in which that machine can be reasoned about. Thus, modules structure the specifications and define a theory structure for reasoning about these properties. The specification structuring features are a significant benefit in the later stages of design and in constructing implementations from detailed designs.

B.1.1 Related Work

A number of other languages have added higher level structuring to Z. In particular, extending Z with object orientation is a popular activity (Object-Z [2], Z++ [13]). One of the main reasons that such an approach was not taken with Sum is that the technology for reasoning about such object-oriented specifications is a subject of ongoing research. In addition, there is no consensus as to appropriate formal refinement methods for the development of object oriented specification. Other approaches to structuring Z specifications [7] are rather more sophisticated than required for development to code (and may make translation or projection to languages such as Ada quite difficult). These approaches tend to have general and powerful notions of structure which admit, for example, partial parametrisation of modules. Again, these approaches do

Squaring

$$sqr, sqrt : \mathbb{N} \to \mathbb{N}$$

$$\forall x : \mathbb{N} \bullet sqr(x) = x * x$$
$$\forall x, k : \mathbb{N} \bullet sqr(k) \leq x \land sqr(k+1) > x \Rightarrow sqrt(x) = k$$

Circle

import Squaring

$$area : \mathbb{N} \to \mathbb{N}$$

$$\forall radius : \mathbb{N} \bullet area(radius) = Squaring.sqr(radius) * 22 \operatorname{div} 7$$

Figure B.1: Simple import.

not have well defined semantics as yet, and support for reasoning about such specifications is only touched upon in their definitions. Conversely, the Sum module concept is relatively simple. Construction of the theories generated from such specification is relatively painless and translation of the modular structure of detailed designs, expressed in Sum, to implementation language equivalents is straightforward.

B.2 The Basic Forms of Modules

Modules provide a means of grouping together and accessing related declarations and definitions. In its simplest form, a module is simply an encapsulation of a sequence of declarations. The chief mechanism for referencing other modules is import. Import brings into scope, but does not make visible, the entities declared in the referenced module. Entities in a referenced module may be accessed via *qualified names*. A qualified name $\alpha.\nu$ consists of a module name α followed by the name of the entity ν defined in α which is to be accessed. If the referenced module imports other modules then these too are in scope and the entities contained therein may also be accessed via qualified names.

B.2.1 Simple Import

Sum provides several forms of import, the first of which is a simple import. An example of this mode of import is given in figure B.1 where the module *Squaring* is imported into the module *Circle*. All the entities in *Squaring* are in scope in module *Circle* and can be accessed by prefixing the name of the entity with the name of the module from which it is exported. Thus, *Squaring.sqr* refers to *sqr* in the module *Squaring*.

Allowing entities in modules to be referenced solely by means of qualified names is excessively cumbersome in practice. To alleviate this problem Sum also provides a visible declaration which makes entities in an imported module directly accessible without the need for a qualified name. It does not make the entities in transitively imported modules visible. Thus in Sum the visible declaration is *non-transitive*. For example, in module *Cylinder* in figure B.2 the addition of the declaration *visible Circle* means that we can reference the entity *area* in *Circle* directly without qualification but we cannot access the entities *sqr* and *sqrt* of *Squaring* directly. *Squaring* is transitively imported into *Cylinder* but is not transitively visible. The entities *sqr* and *sqrt* however, are in scope and may be accessed via qualified names.

Cylinder

import Circle

visible Circle

$$length : \mathbb{N}$$

$$area : \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall radius : \mathbb{N} \bullet area(radius) = 2 * area(radius) + (2 * radius * length * 22) \operatorname{div} 7$$

Figure B.2: Making entities of an imported module visible.

Cylinder′

import Circle

visible Circle

$$length : \mathbb{N}$$

$$area : \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall radius : \mathbb{N} \bullet area(radius) = 2 * Circle.area(radius) + (2 * radius * length * 22) \operatorname{div} 7$$

Figure B.3: Linear visibility.

Note that the majority of the languages which will be the focus of development will have non-transitive reference mechanisms at the module level (cf. Ada, Modula2). Non-transitivity of visible declarations may cause the importation and visibility parts of module definitions to be more verbose than would be the case if all imported items were immediately made visible, but this mechanism does contribute significantly to the narrowing of the module interface. This does not exclude modules from promoting the visibility of an imported entity to the exported interface of the module. Again, this is simply more verbose than would be the case in the transitive counterpart. Enforcing non-transitivity also has implications for theory generation and the associated deduction mechanisms. This is discussed in more detail later.

Visibility of imported entities is *linear*. Given two entities which are in scope and which have the same unqualified name the directly visible entity is the most recently declared. In figure B.3 another entity called *area* is declared in module *Cylinder′*. This entity hides the function *area* already declared in *Circle* even though it is declared to be visible in *Cylinder′*. The function area in *Circle* is still in scope and can be accessed by using its qualified name.

B.2.2 Nesting Modules

Modules may be nested. When a nested module (submodule) is defined, previous declarations of the enclosing module(s) are visible within the submodule. After the declaration of the submodule, the module name is visible, but not the entities declared within the submodule. To make the declared entities of the submodule visible, a visible declaration may be used. Figure B.4 gives an example of a nested module.

Outside the enclosing module, the name of a submodule is brought into scope by importing its enclosing module. However, the entities of the submodule are not directly visible. An explicit visible declaration is required to make the

Circle

Squaring

$$sqr, sqrt : \mathbb{N} \to \mathbb{N}$$

$$\forall x : \mathbb{N} \bullet sqr(x) = x * x$$
$$\forall x, k : \mathbb{N} \bullet sqr(k) \leq x \land sqr(k+1) > x \Rightarrow sqrt(x) = k$$

visible Squaring

$$area : \mathbb{N} \to \mathbb{N}$$

$$\forall radius : \mathbb{N} \bullet area(radius) = sqr(radius) * 22 \text{ div } 7$$

Figure B.4: A nested module.

Circle

import Squaring

visible Squaring{*sqr*}

$$area : \mathbb{N} \to \mathbb{N}$$

$$\forall radius : \mathbb{N} \bullet area(radius) = sqr(radius) * 22 \text{ div } 7$$

Figure B.5: Narrowing a module interface through selective visibility.

submodule entities visible. This provides a convenient means of hiding some details of a module within a nested module.

B.2.3 Narrowing Interfaces and Renaming

Methodologically, the interface to a module should be as narrow as possible. Selectively making some, but not all, components of an imported module visible is possible in Sum by explicitly including in the visibility declaration the entities to be made visible from the referenced module. This is illustrated in figure B.5 where *sqr* in *Squaring* is made visible but *sqrt* is not.

A module *M* may be *instantiated* with a specified name *S*, which declares a new module *S* as a distinct copy of the module *M*. The names of the declared entities of *S* are qualified by the new module name. This is illustrated in figure B.6, which imports *Squaring*, instantiating it as the module *S*. The entities *S.sqr* and *S.sqrt* are available for use. Of course, a *visible S* declaration could be used to permit abbreviated reference to those entities. The implications of instantiation for re-use of state-based specifications is discussed in section B.3.

When a module is instantiated, entities of the module may be re-named. For example, figure B.7 illustrates the importation and instantiation of *Squaring*

Circle

import Squaring as S

$$area : \mathbb{N} \to \mathbb{N}$$

$$\forall radius : \mathbb{N} \bullet area(radius) = S.sqr(radius) * 22 \text{ div } 7$$

Figure B.6: Simple instantiation.

Circle

import Squaring$\{f/sqr\}$ *as S*

visible S

$$area : \mathbb{N} \to \mathbb{N}$$

$$\forall\, radius : \mathbb{N} \bullet area(radius) = f(radius) * 22 \operatorname{div} 7$$

Figure B.7: Instantiation with renaming.

Sort$[Item;\ ord : Item \times Item \to \mathbb{B};\ nord : \mathbb{N} \times \mathbb{N} \to \mathbb{B}]$

$$sorted : \operatorname{seq} Item \to \mathbb{B}$$

$$sorted(\langle\rangle)$$
$$\forall\, L : \operatorname{seq} Item \bullet$$
$$L \neq \langle\rangle \Rightarrow sorted(L) \Leftrightarrow (\forall\, i, j : \operatorname{dom} L \bullet nord(i, j) \Rightarrow ord(L(i), L(j)))$$

$\vdots$

Figure B.8: A parametrised module.

with the entity *sqr* renamed to f.

B.2.4 Parametrisation

Modules may be parametrised with a restricted set of entities. These entities include given sets (types) and operations. For example, Figure B.8 shows part of a sort module parametrised by the type of object to be sorted, the ordering relation on the items to be sorted, and the ordering relation to be used on item indices. Note that all entities within a parametrised module which rely on one of the formal parameters to that module are, in the Z sense, themselves generic.

Instantiation of the module is required for the instantiation of the internal entities. When a parametrised module is referenced, it must be fully instantiated. Partial instantiation is not available in Sum.

Figure B.9 shows the import and instantiation of the *Sort* module with specific parameters. Instantiation and renaming may be combined, as illustrated in figure B.10, where *sorted* is renamed to *natsorted*.

B.3 Support for the Cogito Methodology

In addition to the basic forms of modules described in section B.2, Sum provides constructs which explicitly support the Cogito system specification and program development methodology [1]. In Cogito, systems are specified by defining the set of allowable states, the initial state of the system and the operations which may change the state just like in Z. One difficulty with Z however,

UseSort

import Sort$(\mathbb{N},\ \leq,\ \leq\ )$ *as SortNats*;

$\vdots$

Figure B.9: Instantiating a parametrised module.

UseSort

import Sort$(\mathbb{N},\ \leq,\ \leq\ )$ *as SortNats*[*natsorted*/*sorted*];

$\vdots$

Figure B.10: Instantiating a parametrised module with renaming.

SymTab[*SYM*; *VAL*; $k : \mathbb{N}_1$]

$SType == SYM \twoheadrightarrow VAL$

state

$st : SType$

$\# \operatorname{dom}(st) \leq k$

init

$st' = \{\}$

op lookup

$s? : SYM$

$v! : VAL$

$\operatorname{pre}(s? \in \operatorname{dom} st)$

$v! = st(s?)$

changes_only{}

op update

$s? : SYM$

$v? : VAL$

$\operatorname{pre}(\# \operatorname{dom}(st) < k \land s? \notin \operatorname{dom}(st))$

$st' = st \oplus \{s? \mapsto v?\}$

Figure B.11: Sum specification of simplified symbol table

is that it does not provide an explicit mechanism for stating what roles individual schemas play in the overall specification. Consequently when developing tools to support specification and refinement one Z schema in a specification looks much like another.

Refinement in Cogito occurs between modules and is intended to reflect a refinement of one state machine, specified by a Sum module, by another state machine which is also specified by a Sum module. In formalising the refinement relation it is important to know the role that schemas play in the specification. Consequently, we distinguish between state, initialisation, operation and specification schemas in Sum specifications. The notation presented below is used during the software development process to generate the appropriate proof obligations for refinement. The semantics of schemas however, remains unchanged.

B.3.1 Formalising State Machines

In Sum, a module may define a *state machine*, which consists of a state schema, an initialisation schema, and a number of operation schema. In defining these components, auxiliary schema may be used. Furthermore, Sum allows the statement of explicit preconditions for operations. The explicit precondition is shown separately in the schema notation.

A simple example is shown in Figure B.11 which shows a parametrised module defining a simple symbol table. The state schema is distinguished by being named with the reserved identifier *state* : *Exp*. The initialisation schema is identified by the reserved identifier *init*. The *init* schema implicitly includes a dashed copy of the *state* schema. Thus the *init* schema in full for the example of figure B.11 is:

init

$$st' : SType$$

$$\# \operatorname{dom}(st') \leq k$$
$$st' = \{\}$$

This treatment of initialisation accords with the fact that an initialisation is required to establish the state invariant, but is not entitled to assume it; the 'old' state is not available. Note that we use dashed variables for the initial state here, as in [24]. The motivation for this is that in general an initialisation may need to be refined to the code which establishes it, and the use of the dashed convention allows a uniform treatment of refinement of initialisation and other operations.

The schemas *update* and *Lookup* are identified as operations of the state machine defined by this module by using the keyword *OP*.

An operation schema implicitly includes the *state* schema, and a copy of the *state* schema in which all components are dashed. Thus, in fully explicit form, the operation schema *update* is:

op update

$$s? : SYM$$
$$v? : VAL$$
$$st, st' : SType$$

$$\operatorname{pre}(\# \operatorname{dom}(st) < k \land s? \notin \operatorname{dom}(st))$$
$$st' = st \oplus \{s? \mapsto v?\}$$
$$\# \operatorname{dom}(st) \leq k$$
$$\# \operatorname{dom}(st') \leq k$$

Note that operation schemas have a three-part structure: a signature and two predicates. The first predicate is an explicit precondition. To preserve compatibility with the schema calculus, a schema with explicit precondition is not treated semantically as a new form of schema. Rather, the explicit precondition is conjoined with the rest of the schema. Thus *update* here is equivalent to

op update

$$s? : SYM$$
$$v? : VAL$$
$$st, st' : SType$$

$$\# \operatorname{dom}(st) < k \land s? \notin \operatorname{dom}(st)$$
$$st' = st \oplus \{s? \mapsto v?\}$$
$$\# \operatorname{dom}(st) \leq k$$
$$\# \operatorname{dom}(st') \leq k$$

However, the Cogito toolset generates a proof obligation that the stated precondition, in conjunction with the state invariant, implies the calculated precondition of the schema. Having the precondition expicit is very useful in refinement, since refinement proof obligations often refer to the precondition of an operation.

The notation *changes_only*$\{\}$ in the operation *lookup* asserts that no state variables change. Here the only state variable is the state component st. Thus in this context *changes_only*$(\{\})$ is equivalent to $st' = st$. In general the notation *changes_only*(S) asserts that for all variables v in the signature such that there is a dashed counterpart v' in the signature, and such that v does not occur in S, $v' = v$.

This notation has been introduced to provide a compact and flexible way of asserting that parts of the state remain unchanged. Generally it is more compact to list the variables which may change, as in the *changes_only* predicate, rather than those which do not.

A variant of the *changes_only* notation allows the use of a schema as an argument. If S is a schema, then *changes_only*(S) is equivalent to *changes_only*(V) where V is the set of variables in the signature of S.

B.3.2 Schema Variables and Modules

Schema variables in different modules are defined in different name spaces. The full name of a schema variable v appearing in a module M is $M.v$. The usual visibility mechanisms can be used to enable elision of the module name part. Because schema variables from different modules are distinct, there *cannot* be unintended clashes between schema variables declared in different modules.

Distinctness of schema variables from different modules also means that, as a default, the states of imported module instances are not merged. Consider the example module in figure B.12 which defines a state machine consisting of two symbol tables. It uses one symbol table to store negative key values and the other to store positive key values. Note that the state declaration for *TwoSyms* includes both state schemas from $s1$ and $s2$. Thus the state has access to variables $st1.st$ and $st2.st$.

This kind of apartness for state components is what is most commonly required in building up a composite state from components. In the more unusual situation in which state merging is required, this can be achieved by renaming the schema variables of imported schema to new, common schema variables.

Renaming is used in the operations of figure B.12. For example, the operation *update* uses either $st1.update$ or $s2.update$, depending on whether its argument

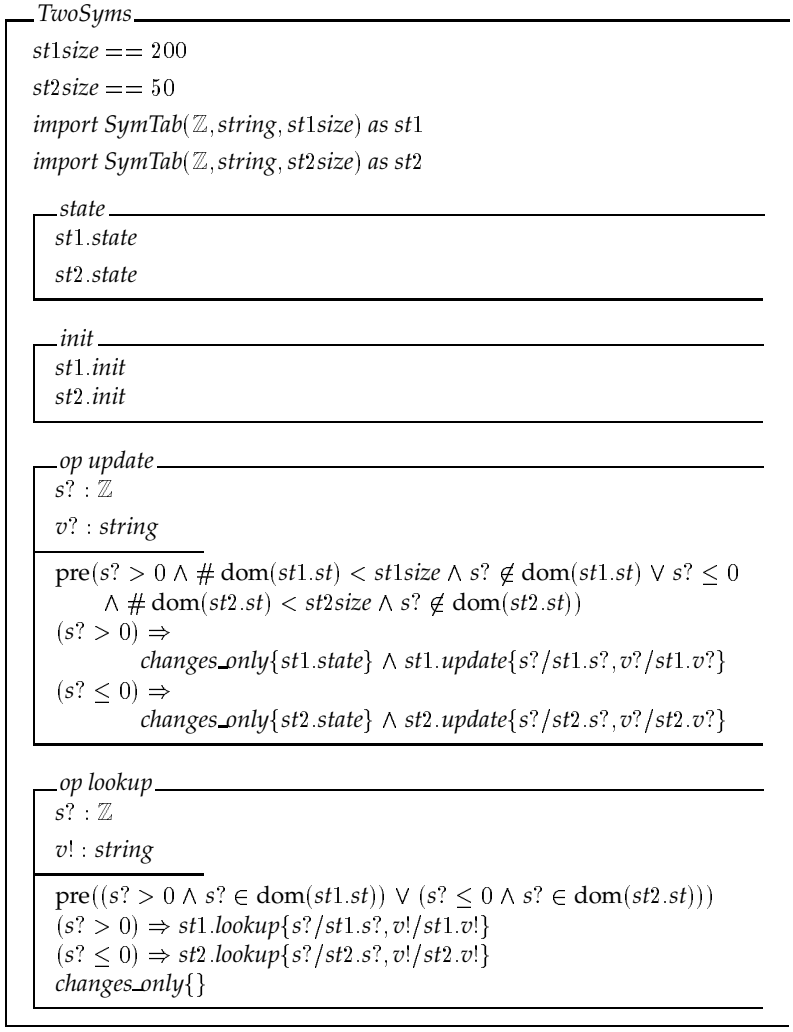


Figure B.12: Sum module defining a state machine with two symbol tables

s? is positive or not. The variables *s*? and *v*? of *update* are new and defined in the current module *TwoSyms*. (Their underlying names are *TwoSyms*.*s*? and *TwoSyms*.*v*?). They are distinct from the variables *st*1.*s*? and *st*1.*v*? of *st*1, for example. Thus to reuse the operation *st*1.*update*, the input variables of *st*1.*update* are renamed to the input variables of *TwoSyms.update*.

Schema variables within one module are defined in the same name space, and thus the schema calculus within a module works as for Z: variables with the same name are merged during schema calculus operations.

B.4 Sum Intermediate Language

B.4.1 Introduction

The Sum intermediate language, (IL), is a way of embedding the semantics of a target programming language within the Z-based formal specification and development language Sum. The embedding uses the 'predicative programming' paradigm for defining the semantics of IL statements as Sum predicates which allows integration with general Sum specification predicates. The approach enables formal tool-supported refinement from specifications to a Sum subset which is then translated to a target programming language. The Sum intermediate language is a subset of the full Sum specification and development language. It can be translated to a variety of imperative languages. Currently automatic translation to Ada is supported. In development by refinement, as supported by Cogito, an abstract specification is transformed step-wise into a form suitable for implementation and, at intermediate stages, development is represented by a mixture of abstract specification and IL code.

Figure B.13 shows a Sum module for a simple symbol table implementation. This module is the result of the refinement process and is in 'pure IL' suitable for translation to executable Ada.

B.4.2 Translation of Sum Modules

symtabimp[*SYM*; *VAL*; $k : natural\_1$]

state

$keys : array(1 \mathinner{..}_c k, SYM)$
$vals : array(1 \mathinner{..}_c k, VAL)$
$high : 0 \mathinner{..}_c k$

$\forall i : 1 \mathinner{..}_c high;\ j : 1 \mathinner{..}_c high \bullet$
$\quad i \neq_c j \Rightarrow keys(i) \neq_c keys(j)$

init

$assign\ \ high := 0$

op update

$s, s' : SYM$
$v, v' : VAL$

$assign\ \ high, keys, vals :=$
$\quad high +_c 1,$
$\quad upd(keys, high +_c 1, s),$
$\quad upd(vals, high +_c 1, v)$

op lookup

$s, s' : SYM$
$v, v' : VAL$

$var\ ind, ind' : 1 \mathinner{..}_c k \bullet$
$\ assign\ \ ind := 1\ ;;$
$\ \ while\ \ keys(ind) \neq_c s\ \ do$
$\ \ \ assign\ \ ind := ind +_c 1\ \ done\ ;;$
$\ assign\ \ v := vals(ind)$

Figure B.13: "symtabimp.sum"

The sum module *symtabimp* (Figure B.13) is translated to a generic Ada package with both a specification and a body (Figure B.14). The translator generates generic Ada packages for both generic and non-generic Sum modules. Generic instantiation in Ada can then be used to implement Sum *import-as*. In addition, non-generic packages are generated for non-generic modules in order to support direct *import* in Sum. On translation to Ada, the module type parameters *SYM* and *VAL* become formal generic private types declared in the package specification. The constant parameter *k* becomes a generic formal object.

Figure B.15 shows the importation of the module *symtabimp* as a new instance *sti*, using *import-as*. Figure B.16 shows the translation of this to package instantiation in Ada. Figure B.15 also illustrates the capability for Sum to be used to model the Ada libraries. As Sum supports generic submodules, the Ada library structure can be directly modeled in Sum allowing, for instance, reference to Ada IO in an IL specification.

Sum provides a *visible* declaration which makes entities in an imported module directly accessible without the need for a qualified name. It does not make the declarations in transitively imported modules visible. For example, it is used in Figure B.15 to make the declarations within *Text_IO* directly visible. The Ada **use** clause broadly corresponds to the Sum *visible* declaration. However, Sum provides *linear visibility*. For example, suppose Sum modules A and B both declare f, and both A and B are imported and made visible in module C. If B is made visible after A then linear visibility determines that $B.f$ is visible in C. In Ada, fully qualified names are required to resolve the resulting name clash.

Returning to the translation of the Sum module in Figure B.13, note that the variables occurring in the *state* schema of the Sum module are translated to public variables in the package specification. The state invariant of the Sum module, guaranteed by the refinement to be preserved by each of the operations, is inserted as a comment to that effect.

IL operation schemas are translated to procedures in the package body (see Figure B.14). The special operation *init* becomes a procedure which is called by the package initialisation part. The only allowable IL statement for initialisation is a (multiple) assignment.

Parameters to IL operations are modeled by pre- and post-state variables using the conventional manner of representing state-change in Z. For example, the parameters to the operation *update* in Figure B.13, s and v, occur both unprimed and primed. In translation, the primed variable is recognized as the post-state version of the unprimed variable, with the result that a single Ada **in out** parameter is generated corresponding to each primed, unprimed pair.

Note that the IL does not use Z-style input variables (decorated with '?') and output variables (decorated with '!') for operation parameters.

Functions

Functions in the IL are purely mathematical (applicative) – see 2.3.10. The body of the function is an intermediate language expression. The Ada functions thus represented cannot affect the state, not be dependent on the state other than through their explicit parameters. A function in the intermediate language corresponds to an axiom with certain restrictions. These restrictions are as follows:

1. The *VarDeclList* must be the declaration of a function whose domain is expressed as either a set or a Cartesian product, and whose range is a set.
2. The *PredList* must be a quantified predicate. The characteristic tuple type of the quantifiable variables must be the same as that of the domain of the function declared in the *VarDeclList*.
3. Within the predicate of the quantified predicate, there must be a predicate of the form

   $$function\_application = expression$$

   where the type of *expression* is that of the range of the domain of the function declared in the *VarDeclList*, and *function_application* is an application of the name of the function declared in the *VarDeclList* to the characteristic tuple of the quantified variables.

With this representation, IL functions have no side effects. Nor is it possible for functions to have imperative bodies or to be implicitly dependent on state.

Example

```
function f : ( integer cross integer ) -|-> integer
pred
    forall a : integer ; b : integer @
        f(a, b) = a + b
end
```

This definition is translated to a function in Ada:

```
function f (a : integer ; b : integer ) is integer
begin
    return a + b
end
```

Package Specification

```
generic
  type SYM is private;
  type VAL is private;
  k: positive;
package gen_symtabimp is
  high : integer range 1 .. k;
  keys : array(1 .. k) of SYM;
  vals : array(1 .. k) of VAL;
  – forall i : 1 ..c high; j :1 ..c high @ i /=c j => keys(i) /=c keys(j)
  procedure init;
  procedure update(s : in out SYM; v:in out VAL);
  procedure lookup(s: in out SYM; v:in out VAL);
end gen_symtabimp;
```

Package Body

```
package body gen_symtabimp is
  procedure init is begin high := 0; end;
  procedure update(s: in out SYM;
                   v: in out VAL) is
  begin
    declare
      high_temp: integer range 0 .. k;
      keys_ind: integer range 1 .. k;
      keys_temp: SYM;
      vals_ind: integer range 1 .. k;
      vals_temp: VAL;
    begin
      high_temp := high + 1;
      keys_ind := high + 1;
      keys_temp := s;
      vals_ind := high + 1;
      vals_temp := v;
      high := high_temp;
      keys(keys_ind) := keys_temp;
      vals(vals_ind) := vals_temp;
    end;
  end;

  procedure lookup(s: in out SYM;
                   v: in out VAL) is
  begin
    declare
      ind: integer range 1 .. k;
    begin
      ind := 1;
      while (keys (ind) /= s) loop
        ind := ind + 1;
      end loop;
      v := vals (ind);
    end ;
  end;
  begin
    init;
end gen_symtabimp;
```

Figure B.14: (Generic) Symtab Implementation

B.4.3 IL data types and their translation

The IL currently includes types which model the following Ada types: boolean, integer, character, string, enumeration types, subrange types, constrained array types and simple record types. The following subsections discuss the modeling of some of these types in Sum and their translation to Ada. The IL does not currently support any modeling of Ada real types, variant record types, access types or derived types.

Computational primitive types

To ensure accurate modeling of implementation and overflow behavior, the IL models operations applicable to Ada's boolean and integer types with specially

defined 'computational' operators rather than Sum's general operators for *int* and *bool* types.

The types *integer*, *natural* and *natural_1* are defined as sub-types of the theoretical (infinite) *int* type and are represented by specific (machine-dependent) ranges of integers.

$$min\_int, max\_int : int$$
$$integer == \{i : int \mid min\_int \leq i \wedge i \leq max\_int\}$$

Machine oriented arithmetic operators are defined so that results are only guaranteed to be defined when results are in range. (Thus they are partial operations on *int*, the theoretical integer type.) For example, $+_c$ is a computational operator specified as follows.

$$+_c : integer \times integer \nrightarrow integer$$
$$\forall x, y : integer \bullet (min\_int \leq x + y \wedge x + y \leq max\_int) \Rightarrow x +_c y = x + y$$

Note that the result of the computational addition is defined only when the (theoretical) sum is within bounds. Integer subranges are modeled by the $..._c$ operator. Other computational arithmetic (and boolean) operators may be characterised similarly.

symtabuser

import Ada; visible Ada
import Text_IO; visible Text_IO
import Integer_IO(integer) as Int_IO
$Sym ::= enum(s1, s2, s3, s4)$
import symtabimp(Sym, integer, 3) as sti

state

sti.state

op op1

$x, x' : integer$
$sx, sx' : Sym$
$y, y' : integer$

call (*sti.update*, $(s => sx, v => x)$, ()) ;;
call (*sti.lookup*, $(s => sx)$, $(v => y)$) ;;
call (*Int_IO.Put*, $(Base => 10, Width => 13, Item => y)$, ())

Figure B.15: "symtabuser.sum"

Enumeration types

In languages such as Ada and Pascal, when an enumeration type is introduced, a number of ordering operators are implicitly introduced. To support reasoning about these operators, a number of declarations and axioms are required. To allow compact representation of enumeration types Sum IL includes specific notation for enumeration types.

An enumerated type is written

$$E ::= enum(C_1, \ldots, C_n)$$

Package Specification

```
with Ada; with Text_IO;
with gen_symtabimp;
package symtabuser is
  use Ada; use Text_IO;
  package Int_IO is new Integer_IO(integer);
  type Sym is (s1, s2, s3, s4);
  package sti is new gen_symtabimp(Sym, integer, 3);
  procedure op1(x : in out integer; sx: in out Sym; y : in out integer);
end symtabuser;
```

Package Body

```
package body symtabuser is
  procedure op1(x : in out integer; sx : in out Sym; y : in out integer) is
  begin
    sti.update(s => sx, v => x);
    sti.lookup(s => sx, v => y);
    Int_IO.Put(Base => 10, Width => 13, Item => y);
  end;
end symtabuser;
```

Figure B.16: Symtab User

and it implicitly declares the type E consisting of the constants $C_1, \ldots C_n$ together with associated operators, *succ*, $<$, etc. For example, Figure B.15 uses an enumerated type to define the type *Sym*. Figure B.16 shows its translation to Ada.

An enumerated type is equivalent to the following definitions in Sum:

$$E ::= C_1 \mid C_2 \mid \ldots \mid C_n;$$
import enum(E) *as E_enum*;
visible E_enum;

where the module *enum* includes declarations for each of the required operators.

enum $[T]$

$$pos : T \to \mathbb{N} \qquad \_ = \_ : T \times T \to \mathbb{B} \qquad \_ \neq \_ : T \times T \to \mathbb{B}$$
$$val : \mathbb{N} \nrightarrow T \qquad \_ < \_ : T \times T \to \mathbb{B} \qquad \_ > \_ : T \times T \to \mathbb{B}$$
$$succ : T \nrightarrow T \qquad \_ \leq \_ : T \times T \to \mathbb{B} \qquad \_ \geq \_ : T \times T \to \mathbb{B}$$

The corresponding Ergo theory includes axioms defining the behavior of the operators.

Arrays

Constrained (fixed-length) array types are represented in the IL by total functions from finite, discrete types to IL types.

An array type, written *array*(*indextype*, *rangetype*) denotes the mathematical type *indextype* $\to$ *rangetype*. Array access is represented by function application. Access out of bounds corresponds to function application off domain, and yields an undefined (error) value. Any such erroneous applications will be discovered during proof of validation and refinement conditions.

Component update is represented by a version of function override, written *upd*, which does not extend the domain of a function.

Note that only 'whole array' update is available in the IL. This provides a simple semantics for array update, enabling the simplification of validation and refinement proofs. However, to achieve greater efficiency, the Ada translator translates these whole array updates to suitable array component updating operations. Translation of assignment, including array assignment is discussed in Section B.4.4.

B.4.4 IL statements and their translation

Predicative programming and the semantics of statements

Sum utilises the usual Z convention for representing state change using primed and unprimed variable names. In general state change is represented by a predicate relating primed and unprimed variable pairs. Moving from Sum specifications to an imperative language requires mapping Sum state change predicates to statements implementing state change in the target imperative language e.g. Fig B.13. The syntax of these statements can be found in 3.3.

The Sum intermediate language includes 'statements' which can be readily translated to corresponding Ada statements. The semantics of these statements is defined by predicates relating primed and unprimed variable pairs . This semantics provides a basis for refining general state change predicates of Sum to IL statements. (See [5] for more details of such a refinement).

For example, the meaning of the IL multiple assignment

$$assign\, x1, \ldots xn := E1, \ldots En$$

where $E1, \ldots En$ are independent of any dashed variables, is (somewhat informally expressed) that

1. $x1' = E1 \wedge \ldots \wedge xn' = En$, and
2. $changes\_only(x1, .., xn)$, i.e., for all variable pairs Y, Y' in the current signature such that Y does not appear on the left of the assignment, $Y' = Y$.

The approach of specifying statement semantics by predicates ('predicative programming') has been investigated by a number of authors (see, for example, [9, 8, 15, 18].) In Cogito we use this approach to specify imperative language statement semantics in the same framework as that of Sum/Z schemas and predicates.

Blocks

We define a local variable block $var\, k, k' : T \bullet B$ as in [18] to mean that

1. for all possible initial k's the precondition of B is true and
2. there exists k, k' such that B is true.

As an example, the *lookup* operation (Figure B.13) introduces a new block with the IL *var*.

Procedure call

Consider the procedure P represented in Sum as follows.

op P

$x1, x1' : T11;\ ...;\ xm, xm' : T1m;\ v1, v1' : T21;\ ...;\ vn, vn' : T2n$

...

A call is represented as follows (see 3.3.4)

$$\textbf{call}\ (P, (x1\ => E1, ..., xm => Em), (v1 => w1, ..., vn => wn))$$

where $E1, ..., Em$ are expressions to be bound to input parameters $x1, .., xm$ and $w1, ..., wn$ are variables to be bound to output parameters $v1, .., vn$. Refer to Fig B.15 for an example.

Finally, to ensure that aliasing problems do not arise, it is required that $w1, ..., wn$ are distinct and do not occur in the signature of P. This constraint ensures that the absence of aliasing is checked during refinement.

Translating IL statements to Ada

Translation of *while* and if_c constructs is straightforward. Here we briefly discuss translation of assignments, blocks and procedure calls.

Multiple assignment To achieve the affect of the Sum multiple assignment in Ada, it is expanded in translation to a sequence of assignments to temporary variables. For example, the Sum multiple assignment in the *update* operation in Figure B.13 has been expanded in the Ada procedure *update* in Figure B.14.

Temporary variables are introduced in the translation in a local **declare** block. Their types are inferred from the declared types of the variables. As seen in the example, array updates are translated using temporary scalar variables of the array index and range types. This relatively simple treatment of translation of array updates is possible because the IL does not support array slices nor may repeated occurrences of the same array appear on the left-hand-side.

The current approach to the translation of multiple assignment is quite conservative. Subsequent analyses, such as data flow analysis, of the expanded assignments would allow further optimisation.

Blocks Sum *var* blocks are translated to Ada **declare** blocks. For example, the *var* block in the *lookup* operation in Figure B.13 is translated to an equivalent Ada **declare** block in Figure B.14. The translator declares the variable *ind* representing the Sum modeling of a variable by *ind* and *ind'*.

Operation call IL operation calls are translated to Ada procedure calls. The operation parameters are translated to Ada named parameter associations. If the input expression is not a simple variable reference, a temporary variable must be used as the actual parameter to the **in out** formal parameter. For example, see the translations of Sum calls in Figure B.15 to corresponding Ada calls in Figure B.16.

Appendix C. Sum declarations for the Mathematical Toolkit

This appendix contains the text of the Sum definitions used as a prelude by the parser/typechecker.

```
module math Is

// predefined base types
[int, string, char, bool];

// Equality Operators
Axiom [T] Is
dec
    _=_         : T cross T --> bool;
    _/=_        : T cross T --> bool
end;

// Relational Operators
_>_  : int cross int --> bool;
_<_  : int cross int --> bool;
_>=_ : int cross int --> bool;
_<=_ : int cross int --> bool;

nat   == { dec x:int | x >= 0 };
nat_1 == { dec x:int | x > 0 };

// Arithmatic Operators
_.._ : int cross int --> power int;
_+_  : int cross int --> int;
_-_  : int cross int --> int;
_*_  : int cross int --> int;
_div_: int cross int --> int;
_mod_: int cross int --> int;
succ : int --> int;
min  : power int --> int;
max  : power int --> int;
_**_ : int cross int --> int;

Axiom [T1, T2] Is
dec
    _|-->_ : T1 cross T2 --> T1 cross T2;
    first  : T1 cross T2 --> T1;
    second : T1 cross T2 --> T2;
    dom_   : power (T1 cross T2) --> power T1;
    ran_   : power (T1 cross T2) --> power T2;
    _dom_restrict_  : (power T1) cross power (T1 cross T2)
                        --> power (T1 cross T2);
    _dom_subtract_  : (power T1) cross power (T1 cross T2)
                        --> power (T1 cross T2);
    _ran_restrict_  : power (T1 cross T2) cross (power T2)
                        --> power (T1 cross T2);
    _ran_subtract_  : power (T1 cross T2) cross (power T2)
                        --> power (T1 cross T2);
    _func_override_ : power (T1 cross T2) cross power (T1 cross T2)
                        --> power (T1 cross T2);
```

```
    inverse: (T1 <--> T2) --> (T2 <--> T1)
end;

Axiom [T1, T2, T3] Is
dec
    _f_compose_: power (T1 cross T2) cross power (T2 cross T3)
                        --> power (T1 cross T3);
    _b_compose_: power (T2 cross T3) cross power (T1 cross T2)
                        --> power (T1 cross T3)
end;

-_          : int --> int;

Axiom [T] Is
dec
    #_          : power T --> nat;
    _^_         : (seq T cross seq T) --> seq T;
    head        : seq T --> T;
    last        : seq T --> T;
    front       : seq T --> seq T;
    tail        : seq T --> seq T;
    rev_        : seq T --> seq T;
    _filter_    : seq T cross power T -->> seq T;
    gen_union_  : power power T --> power T;
    gen_inter_  : power power T --> power T;
    id_         : power T --> power (T cross T);
    _diff_      : (power T cross power T) --> power T;
    _sym_diff_  : (power T cross power T) --> power T;
    _union_     : (power T cross power T) --> power T;
    _inter_     : (power T cross power T) --> power T;
    _in_        : T cross power T --> bool;
    _not_in_    : T cross power T --> bool;
    _subset_    : (power T) cross (power T) --> bool;
    _p_subset_  : (power T) cross (power T) --> bool;
    _in_bag_    : T cross bag T --> bool;
    _bag_union_ : bag T cross bag T --> bag T;
    count       : bag T cross T --> nat;

    iter        : (T <--> T) cross int --> (T <--> T);
    t_closure   : (T <--> T) --> (T <--> T);
    rt_closure  : (T <--> T) --> (T <--> T);
    _partitions_: (seq power T) cross power T --> bool;
    disjoint_   : seq (power T) --> bool
end;

// The computational types integer, natural, and natural_1 are defined as
// subtypes of the theoretical (infinite) int type and are represented by
// a specific (machine dependent) range of integers.
//
integer_first, integer_last : int;

integer == {dec i:int | integer_first <= i and i <= integer_last};

natural_1 == {dec i:integer | i > 0 };

// computational integer operations for IL
// Machine oriented arithmetic operators are defined so that results are
// only guaranteed to be defined when results are within range.
_subc_ : integer cross integer --> integer;
```

```
_addc_  : integer cross integer --> integer;
_multc_  : integer cross integer --> integer;
_divc_: integer cross integer --> integer;
_modc_: integer cross integer --> integer;
_expc_ : integer cross integer --> integer;
_ltc_  : integer cross integer --> bool;
_ltec_ : integer cross integer --> bool;
_gtc_ : integer cross integer --> bool;
_gtec_ : integer cross integer --> bool;
_uptoc_ : integer cross integer --> power integer;
_eqc_ :  integer cross integer  --> bool;
_neqc_ : integer cross integer  --> bool;

// computational boolean operations for IL
_eqc_ : bool  cross bool  --> bool;
_neqc_: bool  cross bool  --> bool;
_andc_ : bool  cross bool  --> bool;
_orc_ : bool  cross bool  --> bool;
_xorc_ : bool  cross bool  --> bool;

// Constrained array types are represented in the IL by total functions
// from finite, discrete types to IL types
axiom [D,R] is
dec
array : (power D) cross (power R) -|-> power (D -|-> R)
end

end math
```

Appendix D. Syntax Tables

These tables summarise the mathematical(Z) and ASCII versions of the symbols used in Sum.

Logic

Mathematical (Z)	ASCII (Sum)
$\Rightarrow$	`=>`
$\wedge$	`and`
$\vee$	`or`
$\Leftrightarrow$	`<=>`
$\neg$	`not`
$\forall$	`forall`
$\bullet$	`@`
$\exists$	`exists`
$\exists_1$	`exists_1`
if..then..else	`if..then..else..fi`

Sets

Mathematical (Z)	ASCII (Sum)
$\varnothing$	`{}`
$\in$	`in`
$\notin$	`not_in`
$\subseteq$	`subset`
$\subset$	`p_subset`
$\mathbb{P}$	`power`
$\mathbb{P}_1$	`power_1`
$\bigcup$	`gen_union`
$\cup$	`union`
$\bigcap$	`gen_inter`
$\cap$	`inter`
$\setminus$	`diff`
$\sim$	`sym_diff`
first	`first`
second	`second`

Finite sets

Mathematical (Z)	ASCII (Sum)
$\mathbb{F}$	`finite`
$\mathbb{F}_1$	`finite_1`
$\#$	`#`

Relations

Mathematical (Z)	ASCII (Sum)
$\leftrightarrow$	`<-->`
$\mapsto$	`|-->`
dom	`dom`
ran	`ran`
$\fatsemi$	`f_compose`
$\circ$	`b_compose`
$\lhd$	`dom_restrict`
$\rhd$	`ran_restrict`
$\ndres$	`dom_subtract`
$\rtimes$	`ran_subtract`
id	`id`
$\sim$	`inverse`
iter	`iter`
$^+$	`t_closure`
*	`rt_closure`

Functions

Mathematical (Z)	ASCII (Sum)
$\pfun$	`-|->`
$\fun$	`-->`
$\pinj$	`>-|->`
$\inj$	`>-->`
$\psurj$	`-|->>`
$\surj$	`-->>`
$\bij$	`>-->>`
$\parallel$	`-||->`
$\inj\!\!\!\!\parallel$	`>-||->`
$\oplus$	`func_override`
λ	`lambda`
μ	`mu`

Sequences

Mathematical (Z)	ASCII (Sum)
$\langle \ldots \rangle$	`< ...>`
seq	`seq`
seq_1	`seq_1`
iseq	`iseq`
$\langle\,\rangle$	`< >`
$\frown$	`concat`
$\frown/$	`d_concat`
head	`head`
tail	`tail`
last	`last`
front	`front`
rev	`rev`
$\upharpoonright$	`filter`
disjoint	`disjoint`
partitions	`partitions`

Bags

Mathematical (Z)	ASCII (Sum)
bag	`bag`
$[\![\ldots]\!]$	`|[...|]`
count	`count`
in_bag	`in_bag`
$\uplus$	`bag_union`
items	`items`

Schemas

Mathematical (Z)	ASCII (Sum)
pre	`pre`
$\upharpoonright$	`^`
$\wedge$	`and`
$\vee$	`or`
$\fatsemi$	`s_compose`
$\setminus$	`\`
/	`/`
Notapplicable	`changes_only`

Computational Integers

Mathematical (Z)	ASCII (Sum)
Notapplicable	`integer` `natural_1`

Computational Arithmetic

Mathematical (Z)	ASCII (Sum)

Notapplicable	`subc`
	`addc`
	`multc`
	`divc`
	`modc`
	`expc`
	`ltc`
	`ltec`
	`gtc`
	`gtec`
	`uptoc`
	`eqc`
	`neqc`

Computational Logic

Mathematical (Z)	ASCII (Sum)

Notapplicable	`eqc`
	`neqc`
	`andc`
	`orc`
	`xorc`

Computational Arrays and Records

Mathematical (Z)	ASCII (Sum)

Notapplicable	`array`
	`upd`
	`is`
	`the`

Computational Statements

Mathematical (Z)	ASCII (Sum)

Notapplicable	`ifc..then..else..fi`
	`while..do..done`
	`assign`
	`call`

Appendix E. Differences between Z and Sum

A comparison of Z and Sum is complicated by the fact that Z is something of a moving target. It has been defined historically by the first and second editions of Spivey's books on *The Z Notation* [19]. A Z standardisation activity is on-going and the result is not guaranteed to match Spivey's definition in all respects. Here we refer primarily to Z as defined in the second edition of Spivey.

Generally speaking, Sum is an extension of Z. Facilities of Sum which extend Z include:

- An ASCII syntax;
- Provision of boolean, string and character data types;
- Facilities for modular specification including parameterised module construction and instantiation;
- Some additional abbreviation mechanisms, such as `changes_only` and `op` annotations on schemas;
- Extra structuring facilities for supporting state machine style specification. These include special syntax/keywords to distinguish the `state`, `init`, and `op` schemas, and the use of a `pre` annotation to identify the precondition of an operation. The identification of these specific roles in Sum specifications allows support for the generation of validation and refinement proof obligations.

There are a few aspects of Z which are not currently supported by Sum. The motivation for these omissions has generally been that either language processing or mechanical proof support for the features was not readily constructable. All of the omissions could be removed in future extensions to the language, and in most cases it is planned to do so.

- The *let* quantifier documented in the second edition of Spivey is not supported.
- Users cannot define their own infix or postfix operators in Sum.
- Abbreviations in Sum require an identifier on the left hand side, whereas Z allows certain patterns. This means, for example, a generic schema cannot be defined by an abbreviation statement.
- In Sum, function application requires bracketing the arguments to the function. Z permits juxtaposition.
- Expressions are not allowed on the left-hand-side in binding selections.
- Sum defines a stricter notion of schema compatibility than Z. Sum requires that the types of the same variable in signatures to be merged be equal, whereas Z requires just base type compatibility. This results in Sum having a simpler semantics for schema operators, which appears to be methodologically desirable.

Further Reading

[1] Anthony Bloesch, Ed Kazmierczak, Peter Kearney, and Owen Traynor. The Cogito Methodology and System. In *Asia Pacific Software Engineering Conference 94*, pages 345–355, 1994.

[2] R. Duke, P. King, G. Rose, and G Smith. The Object-Z Specification Language version 1. Technical Report Technical Report 91-1, Software Verification Research Centre, University of Queensland, 1991.

[3] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[4] Herbert B. Enderton. *Elements of set theory*. Academic Press, New York, 1977.

[5] Nicholas Hamilton, Dan Hazel, Peter Kearney, Owen Traynor, and Luke Wildman. A Complete Formal Development using Cogito. In Chris McDonald, editor, *Computer Science '98*, Australian Computer Science Communications, pages 319–330. Springer-Verlag, 1998.

[6] I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall International Series in Computer Science, 2nd edition, 1993.

[7] Hayes, I.J. and Wildman, L.P. Towards Libraries for Z. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop: Proceedings of the Seventh Annual Z User Meeting*, Workshops in Computing. Springer-Verlag, 1992.

[8] Eric Hehner. *A practical theory of programming*. Springer-Verlag, 1993.

[9] C.A.R. Hoare. Programs are Predicates. *Mathematical Logic and Programming Languages*, pages 141–154, 1985.

[10] Karel Hrbacek and Thomas Jech. *Introduction to Set Theory*. Number 85 in Pure and Applied Mathematics. Dekker, New York, 1984.

[11] E. Kazmierczak. Modularising the specification of a small database system in extended ml. *Formal Aspects of Computing*, 4(1):100–142, 1992. Springer-Verlag.

[12] P. Kearney and L. Wildman. From Formal Specifications to Ada Programs. In *Australian Computer Communications*, pages 193–204, 1999.

[13] K. C. Lano. Z++, an object oriented extension to Z. In J. Nicholls, editor, *Z User Workshop: Proceedings of the Fifth Annual Z User Meeting*, Workshops in Computing. Springer-Verlag, 1991.

[14] Elliott Mendelson. *Introduction to Mathematical Logic*. Mathematics. Wadsworth & Brooks/Cole, Pacific Grove, California, 1987.

[15] Thanh Tung Nguyen. A Relational Model of Demonic Nondeterministic Programs. *Foundations of Computer Science*, 2:101–131, 1991.

[16] John Nicholls, editor. *Z Notation*. Z Standards Panel, ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z), 1995. Version 1.2, ISO Committee Draft; CD 13568.

[17] B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall International Series in Computer Science, 2nd edition, 1996.

[18] E. Sekerinski. A Calculus for Predicative Programming. *Mathematics of Program Construction*, LNCS 669.

[19] J. M. Spivey. *The Z Notation: A Z Reference Manual*. International Series in Computer Science. Prentice Hall, New York, second edition, 1992.

[20] Gaisi Takeuti and W. M. Zaring. *Introduction to Axiomatic Set Theory*. Number 1 in Graduate Texts in Mathematics. Springer-Verlag, New York, 1970.

[21] O. Traynor. The Cogito Repository Manager. In *Asia Pacific Software Engineering Conference 94*, pages 356–365, 1994.

[22] US Government Printing Office. *Reference Manual for the Ada Programming Language - ANSI/MIL-STD 1815A*, 1983.

[23] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[24] J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

Index