Part II
IOA Data Types

IOA specifications can employ various data types, both built-in and user-defined. We list here the operators available for the built-in types; Appendix A defines their properties formally via sets of axioms in multisorted first-order logic (see Section 11). Data types and operators are defined abstractly, not in terms of any particular representation or implementation. In particular, operators are defined without any reference to a "state" or "store," so they cannot have "side-effects."

- The primitive data types `Bool`, `Int`, `Nat`, `Real`, and `Char` can be used without explicit declarations. Section 7 describes the operators available for each of these types.
- Other primitive data types can be introduced as **type** parameters to automaton definitions, as in the channel automaton described in Figure 5, which is parameterized by the types `M` and `Index`.
- Compound data types formed using the type constructors `Array`, `Set`, `Mset`, `Seq`, and `Map` can be used without explicit declarations. Section 8 describes the operators available for these types.
- Compound data types formed using the keywords **enumeration**, **tuple**, and **union** can be used with explicit declarations, as in
  ```
  type Color = enumeration of red, white, blue
  type Msg   = tuple of source, dest: Process, contents: String
  type Fig   = union of sq: Square, circ: Circle
  ```
  Sections 9.8 and 22 describe the operators available for these data types.
- User-defined data types, as well as additional operators on the above primitive and compound data types, can be introduced (or required to have certain properties) by indicating auxiliary specifications, as in
  ```
  axioms RingIndex(abcd, String)
  axioms Stack for Stack[__]
  assumes TotalOrdering(T, <)
  ```
  These auxiliary specifications, which users write as *traits* in the Larch Shared Language (LSL), provide both the syntax and semantics for all operators introduced in this fashion. Sections 9 and 10 describe how to write LSL traits and how to incorporate them into IOA specifications by means of the **axioms** statement.

The equality (__=__), inequality (__≠__), and conditional (**if** __ **then** __ **else**) operators are available for all data types in IOA (the __'s are placeholders for the arguments of these operators).

7 Built-in primitive types

The following built-in primitive types and operators require no declaration.

7.1 Booleans

The boolean data type, `bool`, provides constants and operators for the set $\{true, false\}$ of logical values. Syntactically, the operators $\wedge$ and $\vee$ bind more tightly than $\Rightarrow$, which binds more tightly than $\Leftrightarrow$.

Operators for `bool`	Sample input	Meaning
`true`, `false`	`true`, `false`	The values *true* and *false*
$\neg$	`~p`	Negation (not)
$\wedge$, $\vee$	`p /\ q`, `p \/ q`	Conjunction (and), disjunction (or)
$\Rightarrow$	`p => q`	Implication (implies)
$\Leftrightarrow$	`p <=> q`	Logical equivalence (if and only if)

7.2 Integers

The integer data type, `Int`, provides constants and operators for the set of (positive and negative) integers.

Operators for `Int`	Sample input	Meaning
`0`, `1`, ...	`123`	Non-negative integers
`-`	`-x`	Additive inverse (unary minus)
`abs`	`abs(x)`	Absolute value
`pred`, `succ`	`succ(x)`	Predecessor, successor
`+`, `-`, `*`	`x + (y*z)`	Addition, subtraction, multiplication
`min`, `max`	`min(x, y)`	Minimum, maximum
`div`, `mod`	`mod(x, y)`	Integer quotient, modulus
$<$, $\leq$, $>$, $\geq$	`x <= y`	Less (greater) than (or equal to)

Syntactically, all binary operators bind equally tightly, so that expressions must be parenthesized, as in `((x*y) + z)` $>$ `3`, to indicate the arguments to which operators are applied.

7.3 Natural numbers

The natural number data type, `Nat`, provides constants and operators for the set of non-negative integers. The operators and constants are as for `Int`, except that there are no unary operators `-` or `abs`, there is an additional operator `**` for exponentiation, and the value of `x-y` is defined to be `0` if `x` $<$ `y`. Syntactically, integer constants (e.g., `1`) and operators (e.g., `-`) are distinct from natural number constants and operators that have the same typographical representation. Sometimes such *overloaded* operators can be distinguished from context (e.g., the `1` in the expression `abs(-1)` must be an integer constant, because `abs` and unary `-` are operators over the integers, but not over the natural numbers). At other times, users must distinguish which operators or constants are meant by *qualifying* expressions with types, as in `x` $>$ `0:Nat`.

7.4 Real numbers

The real number data type, `Real`, provides constants and operators for the set of real numbers. Again, the operators and constants are as for `Int`, except that there are no operators `pred`, `succ`, `div`, and `mod`, and there are additional operators `/` and `**` for division and exponentiation.

7.5 Characters

The character data type, `Char`, provides constants and operators for letters and digits.[7]

[7]Additional character constants will be provided in a future version of IOA.

Operators for `Char`	Sample input	Meaning
`'A'`, . . . , `'Z'`, `'a'`, . . . , `'z'`, `'0'`, . . . , `'9'`	`'J'`	Letters and digits
$<, \leq, >, \geq$	`'A' <= 'Z'`	ASCII ordering

7.6 Strings

The string data type, `String`, provides constants and operators for lexicographically ordered sequences of characters. It provides operators as described for `Seq[Char]` (see Section 8.4) as well as the ordering relations $<$, $\leq$, $>$, and $\geq$.

8 Built-in type constructors

The following built-in type constructors and operators require no declaration.

8.1 Arrays

The array data types, `Array[I, E]` and `Array[I, I, E]`, provide constants and operators for one- and two-dimensional arrays of elements of some type `E` indexed by elements of some type `I`.

Operators for `Array[I, E]`	Meaning
`constant(e)`	Array with all elements equal to `e`
`a[i]`	Element indexed by `i` in array `a`
`assign(a, i, e)`	Array `a'` equal to `a` except that `a'[i]` = `e`

Operators for `Array[I, I, E]`	Meaning
`constant(e)`	Array with all elements equal to `e`
`a[i, j]`	Element indexed by `i`, `j` in array `a`
`assign(a, i, j, e)`	Array `a'` equal to `a` except that `a'[i, j]` = `e`

The array (one- or two-dimensional) denoted by `constant(e)` is determined by context, as in `constant(e)[i]`, or by an explicit qualification, as in `constant(e):Array[I,I,E]`.

8.2 Finite sets

The set data type, `Set[E]`, provides constants and operators for finite sets of elements of some type `E`.

Operators for `Set[E]`	Sample input	Meaning
`{}`	`{}`	Empty set
`{...}`	`{e}`	Set containing `e` alone
`insert`	`insert(e, s)`	Set containing `e` and all elements of `s`
`delete`	`delete(e, s)`	Set containing all elements of `s`, but not `e`
∈	`e \in s`	True iff `e` is in `s`
∪, ∩, -	`(s \U s') - (s \I s')`	Union, intersection, difference
⊂, ⊆, ⊃, ⊇	`s \subseteq s'`	(Proper) subset (superset)
`size`	`size(s)`	Size (an `Int`) of `s`

8.3 Multisets

The multiset data type, `Mset[E]`, provides constants and operators for finite multisets of elements of some type `E`. Its operators are those for `Set[E]`, except that there is an additional operator `count` such that `count(e, s)` is the number (an `Int`) of times an element `e` occurs in a multiset `s`.

8.4 Sequences

The sequence data type, `Seq[E]`, provides constants and operators for finite sequences of elements of some type `E`.

Operators for `Seq[E]`	Sample input	Meaning
`{}`	`{}`	Empty sequence
$\vdash$	`s |- e`	Sequence with `e` appended to `s`
$\dashv$	`e -| s`	Sequence with `e` prepended to `s`
$\|\|$	`s || s'`	Concatenation of `s`, `s′`
$\in$	`e \in s`	True iff `e` is in `s`
`head`, `last`	`head(s)`	First (last) element in sequence
`init`, `tail`	`tail(s)`	All but first (last) elements in sequence
`len`	`len(s)`	Length (an `Int`) of `s`
`...[...]`	`s[n]`	`n`th (an `Int`) element in `s`

8.5 Mappings

The mapping data type, `Map[D, R]`, provides constants and operators for finite partial mappings of elements of some domain type `D` to elements of some range type `R`. Finite mappings differ from arrays in two ways: they may not be defined for all elements of `D`, and their domains are always finite.

Operators for `Map[D, R]`	Sample input	Meaning
`empty`	`empty`	Empty mapping
`...[...]`	`m[d]`	Image of `d` under `m`
`defined`	`defined(m, d)`	True if `m[d]` is defined
`update`	`update(m, d, r)`	Mapping `m′` equal to `m` except that `m′[d] = r`

9 Data type semantics

IOA describes the semantics of abstract data types by means of axioms expressed in the the Larch Shared Language (LSL). Users need refer to LSL specifications only if they have questions about the precise mathematical meaning of some operator or if they wish to introduce new operators or data types.[8]

This section provides a tutorial introduction to LSL. It is taken from Chapter 4 of [7], but has been updated to reflect several changes to LSL, most significantly the addition of explicit quantification. LSL is a member of the Larch family of specification languages [7], which supports a two-tiered, definitional style of specification. Each specification has components written in two languages: LSL, which is independent of any programming language, and a so-called *interface language* tailored specifically for a programming language (such as C) or for a mathematical model of

[8]Some tool builders may wish to provide other, equivalent definitions for the built-in data types, e.g., using some other mathematical formalism or in terms of procedures written in some programming language.