Larch Book: Appendix A

July 7, 1993

Contents

Chapter 1

An LSL Handbook

1.1 Introduction

This is the online version of Appendix A of *Larch: Languages and Tools for Formal Specification*, J.V. Guttag, *et al.*, Springer-Verlag, 1993.

READING THE HANDBOOK

This handbook contains a collection of traits written in LSL 2.4 that can be studied to learn more about LSL. Many traits are also suitable for use as specification components. They constitute a library for the LCL and LM3 tools; we hope that they will save others from reinventing wheels—especially polygonal ones. Other traits are more likely to be used as models for the development of similar specialized specification components.

This handbook is representative rather than complete. The LSL tier is open-ended because we believe that no handbook or library will ever include everything that will be needed. Users are encouraged to augment this handbook with additional traits, and to prepare their handbooks for particular applications.

This is not a textbook on discrete mathematics. If you already understand a collection of concepts (e.g., integer arithmetic), their formalization should make sense to you. If you don't, you should still be able to understand precisely what the definitions say (or don't say), but you probably won't get many clues as to why the particular definitions in (say) `Lattice` or `AbelianMonoid` are interesting and useful. Think of this handbook as the "collected formulas" that might appear as an appendix to a mathematics text.

There are many trade-offs in developing this kind of handbook:

- simplicity versus completeness,
- structure (include trait by reference) versus explicitness (copy trait),
- brevity versus explicit indication of consequences,
- concise versus mnemonic names,

- stylistic consistency versus an illustrative range of valid styles,
- standardization (for communication) versus flexibility (for efficiency in particular cases),
- selection among competing notations and definitions for concepts,
- conceptual elegance versus practical utility.

There are general tendencies in the choices exhibited here, but we haven't applied any of our own guidelines slavishly. Many of the stylistic variations are intentional, but there are probably others that we simply didn't notice.

This handbook does not have to be read front-to-back. There is no "correct" order in which to study the traits. Feel free to browse and skip according to your interests and needs. Early sections tend to deal with specific constructs that occur frequently in program interface specifications, while later sections are somewhat more abstract, providing mathematical building blocks that can be used to define, classify, or generalize such constructs. When there didn't seem to be any natural order for things, we fell back on alphabetical order.

Traits in sections labeled *data types* or *data structures* are quite likely to be used directly in interface specifications. Traits in sections labeled *assumptions and implications* or *operator definitions* are more likely to be used in other traits.

Traits are listed in the index. If you don't know exactly what a referenced trait contains, you can always look it up. However, we have tried to use familiar names for familiar concepts. Particularly on first reading, it is probably better to assume that traits such as `Integer` and `TotalOrder` mean what you expect, than to flip continually from trait to trait and section to section.

An `implies` clause does not contribute to the meaning (i.e., the theory) of a legal trait. It is perfectly acceptable to ignore them, and it is often best to do so on first reading. However, they do offer you a chance to check your understanding, by giving examples of facts that are consequences of the definitions in the trait. They may also include alternative (and perhaps more familiar) definitions, or show connections that may not be obvious from looking at just the definitions in the traits.

Both `includes` and `assumes` clauses add axioms from referenced traits. They both have the same semantics within a trait in which they appear, so it's fine to ignore the distinction on first reading. But `assumes` clauses impose an additional proof obligation whenever the trait containing them is referenced in another trait, so they become very relevant when using traits to compose specifications.

Many abstract types are defined in two traits, one of which defines only the essential operators that characterize the type, while the other includes definitions for a richer set of operators in terms of the essential operators. The former kind of trait tends to be used in `assumes` and `implies` clauses; the latter, in `includes` clauses and in interface specifications. Compare, for example, `SetBasics` and `Set`, or `RelationBasics` and `Relation`.

Many traits include `Integer` and use sort `Int` where it might seem that `Natural` and `Nat` would be more natural choices—and, in some cases, would lead to somewhat simpler specifications. This is a consequence of the decision in the interface languages to base all the whole-number types on `Int`. The trait `IntegerPredicates` defines predicates to test for several commonly-used subsets of the integers. The alternative was a large amount of sort-conversion that would severely distract from the clarity of interface specifications. So we pay a small price in the LSL tier for greater simplicity in the interface tier.

If a definition seems “unnatural” to you, you will find it instructive to try to construct a more natural definition yourself. If you find one, you will have gained some experience in writing LSL specifications; if you don’t, you may have gained some insight into the reason for the “unnatural” definition.

The traits in this handbook have passed the scrutiny of the LSL Checker, which parses, expands trait references, resolves overloading, and sort-checks. Most of them have not yet been subjected to semantic checking using LP.

The online version of this handbook is still evolving. The authors would appreciate all kinds of feedback from readers and users. Are there errors or sources of confusion? Have we omitted something that would be widely useful? Are there better ways to define some of the concepts?

NAMING AND LEXICAL CONVENTIONS

Sort names:

- Numeric types: `Int` for integers, `P` for positive numbers, `Q` for rationals, `F` for floating point, and `N` otherwise.
- `T` if there is only one "interesting" sort in the trait.
- Container traits: `E` for elements, `C` for containers.

Operator names:

- $\circ$ for a generic infix operator and also for the composition of maps and relations.
- $\diamond$ for a generic relation.

For convenience in manipulating the online form of the handbook, we have chosen a sequence of ISO Latin characters to represent each non-ISO Latin symbol used in the handbook. Some of them are chosen for visual similarity (e.g., $\rightarrow$ is written as `->` and $\leq$ is written as `<=`); others have been modeled on TeX's choices (e.g., $\circ$ is written as `\circ` and $\in$ is written as `\in`).

Each Larch interface language defines its own notation for literals, based on the programming language's notation; numerical types will generally include the trait schema `DecimalLiterals`.

Many traits have a `size` or `count` operator whose value is always non-negative. For reasons given in the previous section, except within Number theory, we have given their range as `Int`, from trait `Integer`, rather than as `N`, from trait `Natural`.

1.2 Foundations

DATA TYPE: BOOLEAN

```
Boolean: trait
  % This trait is given for documentation only.
  % It is implicit in LSL.
  introduces
    true, false: → Bool
    ¬__: Bool → Bool
    __∧ __, __∨__, __⇒__: Bool, Bool → Bool
  asserts
    Bool generated by true, false
    ∀ b: Bool
      ¬ true == false;
      ¬ false == true;
      true ∧ b == b;
      false ∧ b == false;
      true ∨ b == true;
      false ∨ b == b;
      true ⇒ b == b;
      false ⇒ b == true
  implies
    AC (∧ , Bool),
    AC (∨, Bool),
    Distributive (∨ for +, ∧ for *, Bool for T),
    Distributive (∧ for +, ∨ for *, Bool for T),
    Involutive (¬__, Bool),
    Transitive (⇒ for ◇, Bool for T)
    ∀ b1, b2, b3: Bool
      ¬(b1 ∧ b2) == ¬b1 ∨ ¬b2;
      ¬(b1 ∨ b2) == ¬b1 ∧ ¬b2;
      b1 ∨ (b1 ∧ b2) == b1;
      b1 ∧ (b1 ∨ b2) == b1;
      b2 ∨ ¬b2;
      (b1 = b2) ∨ (b1 = b3) ∨ (b2 = b3);
      b1 ⇒ b2 == ¬b1 ∨ b2
```

Operator definition: if then else

```
Conditional (T): trait
  % This trait is given for documentation only.
  % It is implicit in LSL.
  introduces if__then__else__: Bool, T, T → T
  asserts
    ∀ x, y, z: T
      if true then x else y == x;
      if false then x else y == y
  implies ∀ b: Bool, x: T
    if b then x else x == x
```

1.3 Integers

DATA TYPE

```
Integer (Int): trait
  % The usual (unbounded) integers operators
  includes
    DecimalLiterals (Int for N),
    TotalOrder (Int)
  introduces
    0, 1: → Int
    succ, pred, -__, abs: Int → Int
    __+__, __-__, __*__: Int, Int → Int
    div, mod, min, max: Int, Int → Int
  asserts
    Int generated by 0, succ, pred
    ∀ x, y: Int
      succ(pred(x)) == x;
      pred(succ(x)) == x;
      -0 == 0;
      -succ(x) == pred(-x);
      -pred(x) == succ(-x);
      abs(x) == max(-x, x);
      x + 0 == x;
      x + succ(y) == succ(x + y);
      x + pred(y) == pred(x + y);
      x - y == x + (-y);
      x * 0 == 0;
      x*succ(y) == (x*y) + x;
      x*pred(y) == (x*y) - x;
      y > 0 ⇒ mod(x, y) + (div(x, y) * y) = x;
      y > 0 ⇒ mod(x, y) ≥ 0;
      y > 0 ⇒ mod(x, y) < y;
      min(x, y) == if x ≤ y then x else y;
      max(x, y) == if x ≤ y then y else x;
      x < succ(x)
  implies
    AC (+, Int),
    AC (*, Int),
    AC (min, Int),
    AC (max, Int),
    RingWithUnit (Int for T)
    Int generated by 1, +, -__:Int→Int
    ∀ x, y: Int
      x < y == succ(x) < succ(y);
```

```
    x ≤ y == x < succ(y)
  converts
    1, -__:Int→Int, __-__:Int,Int→Int,
    abs, +, *, div, mod, min, max, ≤, ≥, <, >
```

LITERALS

```
DecimalLiterals (N): trait
  % A built-in trait schema given here
  % for documentation only
  introduces
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 %, ...
      : → N
    succ: N → N
  asserts equations
    1 == succ(0);
    2 == succ(1);
    3 == succ(2);
  % ... as far as needed for any literals
  % of sort N appearing in the including trait
```

OPERATOR DEFINITIONS

```
IntegerPredicates (Int): trait
  % Frequently used subranges of the integers
  assumes Integer
  introduces
    InRange: Int, Int, Int → Bool
    Natural, Positive, Signed, Unsigned: Int → Bool
    maxSigned, maxUnsigned: → Int
  asserts forall n, low, high: Int
    InRange(n, low, high) == low ≤ n ∧ n ≤ high;
    Natural(n) == n ≥ 0;
    Positive(n) == n > 0;
    Signed(n) ==
      InRange(n, -succ(maxSigned), maxSigned);
    Unsigned(n) == InRange(n, 0, maxUnsigned)
  implies ∀ n: Int
    Positive(n) ⇒ Natural(n);
    Unsigned(n) ⇒ Natural(n)
```

1.4 Enumerations

```
Enumeration (T): trait
  % Enumeration, comparison, and ordinal position
  % operators, often used with "enumeration of"
  assumes Integer
  includes DerivedOrders
  introduces
    first, last: → T
    succ, pred: T → T
    ord: T → Int
    val: Int → T
  asserts
    T generated by first, succ
    T generated by last, pred
    ∀ x, y: T
      ord(first) == 0;
      x ≠ last ⇒ ord(succ(x)) = ord(x) + 1;
      x ≠ last ⇒ pred(succ(x)) = x;
      val(ord(x)) == x;
      x ≤ y == ord(x) ≤ ord(y);
      x ≤ last
  implies
    TotalOrder
    T generated by val
    T partitioned by ord
    ∀ x: T
      x ≠ first ⇒ succ(pred(x)) = x;
      x ≠ last ⇒ x < succ(x);
      first ≤ x;
      ord(x) ≥ 0
    converts
      first:→T, succ:T→T, pred:T→T, ord,
        ≤:T,T→Bool, ≥:T,T→Bool,
        <:T,T→Bool, >:T,T→Bool
      exempting succ(last), pred(first)
```

1.5 Containers

Throughout this section we use `E` for the element sort, and `C` for the container sort. This simplifies comparisons among data structures and makes it easier to write generic operator definitions that work for several kinds of containers. Since variable names are local to traits, we imposed no such uniformity on them.

UNORDERED DATA STRUCTURES

```
SetBasics (E, C): trait
  % Essential finite-set operators
  introduces
    { }: → C
    insert: E, C → C
    __∈ __: E, C → Bool
  asserts
    C generated by { }, insert
    C partitioned by ∈
    ∀ s: C, e, e1, e2: E
      ¬(e ∈ { });
      e1 ∈ insert(e2, s) == e1 = e2 ∨ e1 ∈ s
  implies
    InsertGenerated ({ } for empty)
    ∀ e, e1, e2: E, s: C
      insert(e, s) ≠ { };
      insert(e, insert(e, s)) == insert(e, s);
      insert(e1, insert(e2, s)) ==
        insert(e2, insert(e1, s))
    converts ∈
```

```
Set (E, C): trait
  % Common set operators
  includes
    SetBasics,
    Integer,
    DerivedOrders (C, ⊆ for ≤, ⊇ for ≥,
                   ⊂ for <, ⊃ for >)
  introduces
    __∉ __: E, C → Bool
    delete: E, C → C
    {__}: E → C
    __ ∪ __, __ ∩ __, __-__: C, C → C
    size: C → Int
  asserts
    ∀ e, e1, e2: E, s, s1, s2: C
      e ∉ s == ¬(e ∈ s);
      { e } == insert(e, {});
      e1 ∈ delete(e2, s) == e1 ≠ e2 ∧ e1 ∈ s;
      e ∈ (s1 ∪ s2) == e ∈ s1 ∨ e ∈ s2;
      e ∈ (s1 ∩ s2) == e ∈ s1 ∧ e ∈ s2;
      e ∈ (s1 - s2)  == e ∈ s1 ∧ e ∉ s2;
      size({}) == 0;
      size(insert(e, s)) ==
        if e ∉ s then size(s) + 1 else size(s);
      s1 ⊆ s2 == s1 - s2 = {}
  implies
    AbelianMonoid (∪ for ∘, {} for unit, C for T),
    AC (∩, C),
    JoinOp (∪, {} for empty),
    MemberOp ({} for empty),
    PartialOrder (C, ⊆ for ≤, ⊇ for ≥,
                  ⊂ for <, ⊃ for >)
    C generated by {}, {__}, ∪
    ∀ e: E, s, s1, s2: C
      s1 ⊆ s2 ⇒ (e ∈ s1 ⇒ e ∈ s2);
      size(s) ≥ 0
    converts
      ∈, ∉, {__}, delete, size, ∪, ∩, -:C,C→C,
      ⊆, ⊇, ⊂, ⊃
```

```
BagBasics (E, C): trait
  % Essential bag operators
  includes Integer
  introduces
    { }: → C
    insert: E, C → C
    count: E, C → Int
  asserts
    C generated by { }, insert
    C partitioned by count
    ∀ b: C, e, e1, e2: E
      count(e, { }) == 0;
      count(e1, insert(e2, b)) ==
        count(e1, b) + (if e1 = e2 then 1 else 0)
  implies
    InsertGenerated ({ } for empty)
    ∀ e: E, b: C
      insert(e, b) ≠ { };
      count(e, b) ≥ 0
    converts count
```

```
Bag (E, C): trait
  % Common bag operators
  includes
    BagBasics,
    DerivedOrders (C, ⊆ for ≤, ⊇ for ≥,
                   ⊂ for <, ⊃ for >)
  introduces
    delete: E, C → C
    {__}: E → C
    __∈ __, __∉ __: E, C → Bool
    size: C → Int
    __∪ __, __-__: C, C → C
  asserts
    ∀ e, e1, e2: E, b, b1, b2: C
      count(e1, delete(e2, b)) ==
        if e1 = e2 then max(0, count(e1, b) - 1)
        else count(e1, b);
      { e } == insert(e, { });
      e ∈ b == count(e, b) > 0;
      e ∉ b == count(e, b) = 0;
      size({ }) == 0;
      size(insert(e, b)) == size(b) + 1;
      count(e, b1 ∪ b2) ==
        count(e, b1) + count(e, b2);
      count(e, b1 - b2) ==
        max(0, count(e, b1) - count(e, b2));
      b1 ⊆ b2 == b1 - b2 = { };
  implies
    AbelianMonoid (∪ for ∘, { } for unit, C for T),
    JoinOp (∪, { } for empty),
    MemberOp ({ } for empty),
    PartialOrder (C, ⊆ for ≤, ⊇ for ≥,
                  ⊂ for <, ⊃ for >)
    ∀ e, e1, e2: E, b, b1, b2: C
      insert(e, b) ≠ { };
      count(e, b) ≥ 0;
      count(e, b) ≤ size(b);
      b1 ⊆ b2 ⇒ count(e, b1) ≤ count(e, b2)
    converts count, ∈, ∉, {__}, ∪, -:C,C→C,
      delete, size, ⊆, ⊇, ⊂, ⊃
```

INSERTION ORDERED DATA STRUCTURES

```
StackBasics (E, C): trait
  % Essential LIFO operators
  includes Integer
  introduces
    empty: → C
    push: E, C → C
    top: C → E
    pop: C → C
  asserts
    C generated by empty, push
    ∀ e: E, stk: C
      top(push(e, stk)) == e;
      pop(push(e, stk)) == stk;
  implies converts top, pop
    exempting top(empty), pop(empty)

Stack (E, C): trait
  % Common LIFO operators
  includes StackBasics, Integer
  introduces
    count: E, C → Int
    __ ∈ __: E, C → Bool
    size: C → Int
    isEmpty: C → Bool
  asserts
    ∀ e: E, stk: C
      size(empty) == 0;
      size(push(e, stk)) == size(stk) + 1;
      isEmpty(stk) == stk = empty
  implies
    Container (push for insert, top for head,
               pop for tail)
    C partitioned by top, pop, isEmpty
    ∀ stk: C
      size(stk) ≥ 0
    converts top, pop, count, ∈, size, isEmpty
      exempting top(empty), pop(empty)
```

```
Queue (E, C): trait
  % FIFO operators
  includes Integer
  introduces
    empty: → C
    append: E, C → C
    count: E, C → Int
    __ ∈ __: E, C → Bool
    head: C → E
    tail: C → C
    len: C → Int
    isEmpty: C → Bool
  asserts
    C generated by empty, append
    ∀ q: C, e, e1: E
      count(e, empty) == 0;
      count(e, append(e1, q)) ==
        count(e, q) + (if e = e1 then 1 else 0);
      e ∈ q == count(e, q) > 0;
      head(append(e, q)) ==
        if q = empty then e else head(q);
      tail(append(e, q)) ==
        if q = empty then empty
        else append(e, tail(q));
      len(empty) == 0;
      len(append(e, q)) == len(q) + 1;
      isEmpty(q) == q = empty
  implies
    Container (append for insert)
    C partitioned by head, tail, isEmpty
    ∀ q: C
      len(q) ≥ 0
    converts head, tail, len
      exempting head(empty), tail(empty)
```

```
Deque (E, C): trait
  % Double ended queue operators
  includes Integer
  introduces
    empty: → C
    __ ⊣ __: E, C → C
    __ ⊢ __: C, E → C
    count: E, C → Int
    __ ∈ __: E, C → Bool
    head, last: C → E
    tail, init: C → C
    len: C → Int
    isEmpty: C → Bool
  asserts
    C generated by empty, ⊢
    ∀ e, e1, e2: E, d: C
      count(e, empty) == 0;
      count(e, e1 ⊣ d) ==
        count(e, d) + (if e = e1 then 1 else 0);
      e ∈ d == count(e, d) > 0;
      e ⊣ empty == empty ⊢ e;
      (e1 ⊣ d) ⊢ e2 == e1 ⊣ (d ⊢ e2);
      head(e ⊣ d) == e;
      last(d ⊢ e) == e;
      tail(e ⊣ d) == d;
      init(d ⊢ e) == d;
      len(empty) == 0;
      len(d ⊢ e) == len(d) + 1;
      isEmpty(d) == d = empty
  implies
    Stack (head for top, tail for pop,
           ⊣ for push, len for size),
    Queue (⊣ for append, last for head,
           init for tail)
    C generated by empty, ⊣
    C partitioned by len, head, tail
    C partitioned by len, last, init
    ∀ d: C
      d ≠ empty
        ⇒ (head(d) ⊣ tail(d) = d
            ∧ init(d) ⊢ last(d) = d)
    converts head, last, tail, init, len
      exempting head(empty), last(empty),
        tail(empty), init(empty)
```

```
List (E, C): trait
  % Add singleton and concatenation
  includes Deque
  introduces
    {__}: E → C
    __ || __: C, C → C
  asserts ∀ e: E, ls, ls1, ls2: C
    {e} == empty ⊢ e;
    ls || empty == ls;
    ls1 || (ls2 ⊢ e) == (ls1 || ls2) ⊢ e
  implies
    C generated by empty, {__}, ||
    converts head, last, tail, init, len, {__}, ||
      exempting head(empty), last(empty),
        tail(empty), init(empty)

String (E, C): trait
  % Index, substring
  includes List
  introduces
    __[__]: C, Int → E
    prefix: C, Int → C
    removePrefix: C, Int → C
    substring: C, Int, Int → C
  asserts ∀ e: E, s: C, i, n: Int
    tail(empty) == empty;
    init(empty) == empty;
    s[0] == head(s);
    n ≥ 0 ⇒ s[n + 1] = tail(s)[n];
    prefix(empty, n) == empty;
    prefix(s, 0) == empty;
    n ≥ 0
      ⇒ prefix(e ⊣ s, n + 1) = e ⊣ prefix(s, n);
    removePrefix(s, 0) == s;
    n ≥ 0
      ⇒ removePrefix(s, n + 1)
          = removePrefix(tail(s), n);
    substring(s, 0, n) == prefix(s, n);
    i ≥ 0
      ⇒ substring(s, i + 1, n)
          = substring(tail(s), i, n)
  implies
    IndexOp (⊣ for insert)
    C partitioned by len, __[__]
    converts tail, init
```

```
Sequence (E, C): trait
  % Comparison, subsequences
  assumes StrictPartialOrder (>, E)
  includes
    LexicographicOrder,
    String
  introduces
    isPrefix, isSubstring, isSubsequence: C, C → Bool
    find: C, C → Int
  asserts ∀  e, e1, e2: E, s, s1, s2: C
    isPrefix(s1, s2) == s1 = prefix(s2, len(s1));
    isSubstring(s1, s2) ==
      isPrefix(s1, s2) ∨ isSubstring(s1, tail(s2));
    isSubsequence(empty, s);
    ¬isSubsequence(e ⊣ s, empty);
    isSubsequence(e1 ⊣ s1, e2 ⊣ s2) ==
      (e1 = e2 ∧ isSubsequence(s1, s2))
        ∨ isSubsequence(e1 ⊣ s1, s2);
    find(empty, empty) == 0;
    find(s1, e -| s2) ==
      if isPrefix(s1, e -| s2) then 0
      else find(s1, s2) + 1
  implies
    IsPO (isPrefix, C),
    IsPO (isSubstring, C),
    IsPO (isSubsequence, C)
    ∀ s, s1, s2: C, i, n: Int
      isPrefix(prefix(s, n), s);
      isSubstring(substring(s, i, n), s);
      isSubstring(s1, s2) ⇒ isSubsequence(s1, s2)
    converts
        isPrefix, isSubstring, isSubsequence, find
      exempting ∀ s: C, e: E  find(e ⊣ s, empty)
```

CONTENT ORDERED DATA STRUCTURES

```
PriorityQueue (>:E,E→Bool, E, C): trait
  % Enumerate by order on elements
  assumes TotalOrder (E for T)
  includes Integer
  introduces
    empty: → C
    add: E, C → C
    count: E, C → Int
    __ ∈ __: E, C → Bool
    head: C → E
    tail: C → C
    len: C → Int
    isEmpty: C → Bool
  asserts
    C generated by empty, add
    C partitioned by head, tail, isEmpty
    ∀ e, e1: E, q: C
      count(e, empty) == 0;
      count(e, add(e1, q)) ==
        count(e, q) + (if e = e1 then 1 else 0);
      e ∈ q == count(e, q) > 0;
      head(add(e, q)) ==
        if q = empty ∨ e > head(q) then e
        else head(q);
      tail(add(e, q)) ==
        if q = empty ∨ e > head(q) then q
        else add(e, tail(q));
      len(empty) == 0;
      len(add(e, q)) == len(q) + 1;
      isEmpty(q) == q = empty
  implies
    Container (add for insert)
    ∀ e, e1, e2: E, q: C
      add(e1, add(e2, q)) = add(e2, add(e1, q));
      len(q) ≥ 0;
      add(e, q) ≠ empty
    converts count, ∈, head, tail, len, isEmpty
      exempting head(empty), tail(empty)
```

```
ChoiceSet (E, C): trait
  % A set with a weakly-specified choose operator
  includes Set
  introduces
    choose: C → E
    rest: C → C
    isEmpty: C → Bool
  asserts ∀ e, e1: E, s: C
    s ≠ { } ⇒ choose(s) ∈ s;
    s ≠ { } ⇒ rest(s) = delete(choose(s), s);
    isEmpty(s) == s = { }
  implies
    C partitioned by choose, rest, isEmpty
    ∀ e: E, s: C
      s ≠ { } ⇒ s = insert(choose(s), rest(s))
```

```
ChoiceBag (E, C): trait
  % A bag with a weakly-specified choose operator
  includes Bag
  introduces
    choose: C → E
    rest: C → C
    isEmpty: C → Bool
  asserts ∀ e, e1: E, b: C
    b ≠ { } ⇒ choose(b) ∈ b;
    b ≠ { } ⇒ rest(b) = delete(choose(b), b);
    isEmpty(b) == b = { }
  implies
    Container (choose for head, rest for tail,
               { } for empty)
    C partitioned by choose, rest, isEmpty
    ∀ e: E, b: C
      b ≠ { } ⇒ b = insert(choose(b), rest(b))
```

ASSUMPTIONS AND IMPLICATIONS

```
InsertGenerated (E, C): trait
  % C's contain finitely many E's
  introduces
    empty: → C
    insert: E, C → C
  asserts
    C generated by empty, insert

Container (E, C): trait
  % head and tail enumerate contents of a C
  includes InsertGenerated, Integer
  introduces
    isEmpty: C → Bool
    count: E, C → Int
    __ ∈ __: E, C → Bool
    head: C → E
    tail: C → C
  asserts
    C partitioned by isEmpty, head, tail
    ∀ e, e1: E, c: C
      isEmpty(empty);
      ¬isEmpty(insert(e, c));
      count(e, empty) == 0;
      count(e, insert(e1, c)) ==
        count(e, c) + (if e = e1 then 1 else 0);
      e ∈ c == count(e, c) > 0;
      ¬isEmpty(c) ⇒
        count(e, insert(head(c), tail(c)))
          = count(e, c)
  implies
    ∀ c: C
      ¬isEmpty(c) ⇒ count(head(c), c) > 0;
    converts isEmpty, count, ∈
```

OPERATOR DEFINITIONS

```
MemberOp: trait
  assumes InsertGenerated
  introduces
    __ ∈ __, __ ∉ __: E, C → Bool
  asserts ∀ e, e1, e2: E, c: C
    e ∉ c == ¬(e ∈ c);
    e ∉ empty;
    e1 ∈ insert(e2, c) == e1 = e2 ∨ e1 ∈ c
  implies converts ∈, ∉

JoinOp (⋈): trait
  % Container combining operator
  % e.g., union, concatenation
  assumes InsertGenerated
  introduces __⋈ __: C, C → C
  asserts ∀ e: E, c, c1, c2: C
    empty ⋈ c == c;
    insert(e, c1) ⋈ c2 == insert(e, c1 ⋈ c2)
  implies
    Associative (⋈, C)
    converts ⋈

ReverseOp: trait
  % An operator on lists commonly used
  % to demonstrate theorem provers.
  assumes List
  introduces reverse: C → C
  asserts ∀ e: E, l, l1, l2: C
    reverse(empty) == empty;
    reverse(e ⊣ l) == reverse(l) ⊢ e
  implies ∀ e: E, l, l1, l2: C
      reverse(reverse(l)) == l;
      l ≠ empty ⇒ head(reverse(l)) = last(l);
      l ≠ empty
        ⇒ tail(reverse(l)) = reverse(init(l));
      len(reverse(l)) == len(l);
      reverse(l1 ∥ l2) == reverse(l2) ∥ reverse(l1)
    converts reverse
```

```
IndexOp: trait
  % Select the i-th  element in the container
  % (in enumeration order).
  assumes Integer, Container
  introduces __[__]: C, Int → E
  asserts ∀ c: C, i: Int
    c[0] == head(c);
    i ≥ 0 ⇒ c[i+1] = tail(c)[i]
```

`CoerceContainer (DC, RC)` defines an operator to convert from a term of one container sort, `DC`, to a term of another container sort, `RC`, with the same elements inserted in the same order. For example, a stack can be mapped to a queue. More interestingly, a list can be mapped to a bag, or a bag to a set; these mappings lose information on order and on multiplicity, respectively, so the inverse mappings would introduce inconsistencies.

```
CoerceContainer (DC, RC): trait
  % Insert each element of DC in a new RC
  assumes
    InsertGenerated (DC for C),
    InsertGenerated (RC for C)
  introduces coerce: DC → RC
  asserts ∀ dc: DC, e: E
    coerce(empty) == empty;
    coerce(insert(e, dc)) == insert(e, coerce(dc))
  implies converts coerce
```

```
Permutation (E, C): trait
  % Test for having the same elements
  assumes Container
  includes
    Bag (B for C),
    CoerceContainer (C for DC, B for RC)
  introduces isPermutation: C, C → Bool
  asserts ∀ c1, c2: C
      isPermutation(c1, c2) == coerce(c1) = coerce(c2)
  implies ∀ e: E, c1, c2: C
    isPermutation(c1, c2)
      ⇒ count(e, c1) = count(e, c2)
```

The following traits "promote" various operators on elements to corresponding operators on containers.

```
ElementTest (pass, E, C, T): trait
  % filter collects elements accepted by pass
  assumes InsertGenerated
  introduces
    pass: E, T → Bool
    filter: C, T → C
    allPass: C, T → Bool
    somePass: C, T → Bool
  asserts ∀ c: C, e: E, t: T
    filter(empty, t) == empty;
    filter(insert(e, c), t) ==
       if pass(e, t) then insert(e, filter(c, t))
       else filter(c, t);
    allPass(empty, t);
    allPass(insert(e, c), t) ==
      pass(e, t) ∧ allPass(c, t);
    somePass(c, t) == filter(c, t) ≠ empty
  implies converts filter, somePass, allPass

PairwiseExtension (∘, ⊙, E, C): trait
  % Induce a binary operator on containers
  % from a binary operator on elements.
  assumes Container (E, C)
  introduces
    __∘ __: E, E → E
    __⊙ __: C, C → C
  asserts ∀ e1, e2: E, c1, c2: C
    empty ⊙ empty == empty;
    (c1 ≠ empty ∧ c2 ≠ empty)
      ⇒ c1 ⊙ c2 = insert(head(c1) ∘ head(c2),
                         tail(c1) ⊙ tail(c2));
  implies converts ⊙
    exempting ∀ e: E, c: C
      empty ⊙ insert(e, c), insert(e, c) ⊙ empty
```

```
PointwiseImage: trait
  % Apply elemOp to each element
  assumes
    InsertGenerated (DE for E, DC for C),
    InsertGenerated (RE for E, RC for C)
  introduces
    elemOp: DE → RE
    containerOp: DC → RC
  asserts ∀ dc: DC, de: DE
    containerOp(empty) == empty;
    containerOp(insert(de, dc)) ==
      insert(elemOp(de), containerOp(dc))
  implies converts containerOp

ReduceContainer (unit, ∘): trait
  % Insert the operator in enumeration order.
  assumes Container
  introduces
    unit: → E
    __ ∘ __: E, E → E
    reduce: C → E
  asserts ∀ c: C
    reduce(c) ==
      if c = empty then unit
      else head(c) ∘ reduce(tail(c))
  implies converts reduce
```

1.6 Branching structures

DATA STRUCTURES

The following trait defines the operators on a list (of sort `C`), each of whose elements (of sort `E`) is either an atom (of sort `A`) or a list.

```
ListStructure (A, E, C): trait
  % Classical LISP
  includes List
  E union of list: C, atom: A
```

```
BinaryTree (E, T): trait
  % One of the many interesting tree structures
  introduces
    [__]: E → T
    [__, __]: T, T → T
    content: T → E
    first, second: T → T
    isLeaf: T → Bool
  asserts
    T generated by [__], [__, __]
    T partitioned by content, first, second, isLeaf
    ∀ e: E, t1, t2: T
      content([e]) == e;
      first([t1, t2]) == t1;
      second([t1, t2]) == t2;
      isLeaf([e]);
      ¬isLeaf([t1, t2])
  implies converts isLeaf
```

OPERATOR DEFINITIONS

```
ListStructureOps (A, E, C): trait
  % Operators frequently used in
  % theorem proving demonstrations.
  assumes ListStructure
  introduces
    flatten, reverseAll: C → C
    countAtoms: C → Int
  asserts ∀ a: A, l, l1, l2: C
    flatten(empty) == empty;
    flatten(atom(a) ⊣ l) == atom(a) ⊣ flatten(l);
    flatten(list(l1) ⊣ l2) ==
      flatten(l1) ‖ flatten(l2);
    reverseAll(empty) == empty;
    reverseAll(atom(a) ⊣ l) ==
      reverseAll(l) ⊢ atom(a);
    reverseAll(list(l1) ⊣ l2) ==
      reverseAll(l2) ⊢ list(reverseAll(l1));
    countAtoms(l) == len(flatten(l))
  implies
    ∀ l, l1, l2: C
      flatten(l1 ‖ l2) == flatten(l1) ‖ flatten(l2);
      flatten(flatten(l)) == flatten(l);
      reverseAll(l1 ‖ l2) ==
        reverseAll(l2) ‖ reverseAll(l1);
      reverseAll(flatten(l)) ==
        flatten(reverseAll(l));
      reverseAll(reverseAll(l)) == l;
      countAtoms(l1 ‖ l2) ==
        countAtoms(l1) + countAtoms(l2);
      countAtoms(flatten(l)) == countAtoms(l);
      countAtoms(reverseAll(l)) == countAtoms(l)
    converts flatten, reverseAll, countAtoms
```

1.7 Maps

DATA STRUCTURES

Arrays are heavily-used data structures; programming languages often provide a large number of operators. The following definitions are only a sample.

```
Array1 (E, I, A): trait
  % Basic one-dimensional array operators
  introduces
    assign: A, I, E → A
    __[__]: A, I → E
  asserts
    ∀ a: A, i, j: I, e: E
      assign(a, i, e)[j] ==
        if i = j then e else a[j]
```

```
Array2 (E, I1, I2, A): trait
  % Basic two-dimensional array operators
  introduces
    assign: A, I1, I2, E → A
    __[__, __]: A, I1, I2 → E
  asserts
    ∀ a: A, i1, j1: I1, i2, j2: I2, e: E
      assign(a, i1, i2, e)[j1, j2] ==
        if i1 = j1 ∧ i2 = j2 then e else a[j1, j2]
```

```
ArraySlice2 (E, I1, I2, A): trait
  % A two-dimensional array
  % treated as a vector of vectors
  includes
    Array1 (E, I2, A1),
    Array1 (A1, I1, A)
  introduces
    assign: A, I1, I2, E → A
    __[__, __]: A, I1, I2 → E
  asserts
    ∀ a: A, i1: I1, i2: I2, e: E
      a[i1, i2] == (a[i1])[i2];
      assign(a, i1, i2, e) ==
        assign(a, i1, assign(a[i1], i2, e))
```

The maps of the following trait are finitely generated by {} and `update`.

```
FiniteMap (M, D, R): trait
  % An M is a map from D’s to R’s.
  introduces
    { }: → M
    update: M, D, R → M
    apply: M, D → R
    defined: M, D → Bool
  asserts
    M generated by { }, update
    M partitioned by apply, defined
    ∀ m: M, d, d1, d2: D, r: R
      apply(update(m, d2, r), d1) ==
        if d1 = d2 then r else apply(m, d1);
      ¬defined({ }, d);
      defined(update(m, d2, r), d1) ==
        d1 = d2 ∨ defined(m, d1)
  implies
    Array1 (update for assign, apply for __[__],
            M for A, D for I, R for E)
    converts apply, defined
      exempting ∀ d: D apply({ }, d)
```

OPERATOR DEFINITION

```
ComposeMaps (M1, M2, D, T, R): trait
  % If m1 is a map from D to T
  % and m2 is a map from T to R,
  % m1 ∘ m2 is a map from D to R.
  assumes
    FiniteMap (M1, T, R),
    FiniteMap (M2, D, T)
  includes FiniteMap
  introduces __ ∘ __: M1, M2 → M
  asserts ∀ m1: M1, m2: M2, d: D
    apply(m1 ∘ m2, d) == apply(m1, apply(m2, d));
    defined(m1 ∘ m2, d) ==
      defined(m2, d) ∧ defined(m1, apply(m2, d))
```

1.8 Relations

DATA STRUCTURE

The following traits do not presume that the domain sort, `E`, is generated by any fixed set of operators. Subsets of `E` are represented by subrelations of the identity relation.

```
Relation (E, R): trait
  includes
    RelationBasics,
    RelationOps,
    RelationPredicates

RelationBasics (E, R): trait
  % e1 ⟨ r ⟩ e2 means e1 is related to e2 by r.
  introduces
    __ ⟨ __ ⟩ __: E, R, E → Bool
    ⊥, ⊤, I: → R
    [__, __]: E, E → R
    -__, __⁻¹: R → R
    __ ∪ __: R, R → R
  asserts
    R partitioned by __ ⟨ __ ⟩ __
    ∀ e, e1, e2, e3, e4: E, r, r1, r2: R
      ¬(e1 ⟨ ⊥ ⟩ e2);
      e1 ⟨ ⊤ ⟩ e2;
      e1 ⟨ I ⟩ e2 == e1 = e2;
      e1 ⟨ [e2, e3] ⟩ e4 == e1 = e2 ∧ e3 = e4;
      e1 ⟨ -r ⟩ e2 == ¬(e1 ⟨ r ⟩ e2);
      e1 ⟨ r⁻¹ ⟩ e2 == e2 ⟨ r ⟩ e1;
      e1 ⟨ r1 ∪ r2 ⟩ e2 == e1 ⟨ r1 ⟩ e2 ∨ e1 ⟨ r2 ⟩ e2
  implies
    AbelianMonoid (⊥ for unit, ∪ for ∘, R for T),
    Involutive (__⁻¹, R),
    Involutive (-__, R)
    equations
      -⊥ == ⊤;
      -⊤ == ⊥;
      ⊥⁻¹ == ⊥;
      ⊤⁻¹ == ⊤
    converts ∪, -__, __⁻¹
```

OPERATOR DEFINITIONS

The `skolem` operator is introduced solely to get around the absence of existential quantifiers in LSL.

```
RelationOps: trait
  % Useful non-primitive operators on relations.
  assumes RelationBasics
  includes
    DerivedOrders (R, ⊆ for ≤, ⊇ for ≥,
                   ⊂ for <, ⊃ for >)
  introduces
    __ ∈ __, __ ∉ __: E, R → Bool
    set, dom, range, __⁺, __*: R → R
    __ ∩ __, __ ∘ __, __ - __, __ × __: R, R → R
    domRestrict, rangeRestrict, image: R, R → R
    skolem: E, R, R, E → E
  asserts
    ∀ e, e1, e2, e3: E, r, r1, r2: R
      e ∈ r == e ⟨ r ⟩ e;
      e ∉ r == ¬(e ∈ r);
      set(r) == r ∩ I;
      dom(r) == set(r ∘ ⊤);
      range(r) == set(⊤ ∘ r);
      e1 ⟨ r1 ∩ r2 ⟩ e2 == e1 ⟨ r1 ⟩ e2 ∧ e1 ⟨ r2 ⟩ e2;
      (e1 ⟨ r1 ⟩ e2 ∧ e2 ⟨ r2 ⟩ e3)
        ⇒ e1 ⟨ r1 ∘ r2 ⟩ e3;
      e1 ⟨ r1 ∘ r2 ⟩ e2
        ⇒ (e1 ⟨ r1 ⟩ skolem(e1, r1, r2, e2)
           ∧ skolem(e1, r1, r2, e2) ⟨ r2 ⟩ e2);
      r⁺ == r ∘ (r*);
      r* == I ∪ (r⁺);
      (r1 = I ∪ r2 ∧ r2 = r ∘ r1) ⇒
        ((r*) ⊆ r1 ∧ (r⁺) ⊆ r2);
      r1 - r2 == r1 ∩ (-r2);
      r1 × r2 == set(r1) ∘ ⊤ ∘ set(r2);
      r1 ⊆ r2 == r1 - r2 = ⊥;
      domRestrict(r1, r2) == r1 ∩ (r2 ∘ ⊤);
      image(r1, r2) == set(r1) ∘ r2;
      rangeRestrict(r1, r2) == r1 ∩ (⊤ ∘ r2)
  implies
    AbelianMonoid (⊤ for unit, ∩ for ∘, R for T),
    Distributive (∪ for +, ∩ for *, R for T),
    Distributive (∩ for +, ∪ for *, R for T),
    Idempotent (set, R),
```

```
    Monoid (I for unit, R for T),
    Lattice (R for T, ∪ for ⊔, ∩ for ⊓,
      ⊆ for ≤, ⊇ for ≥, ⊂ for <, ⊃ for >),
    PartialOrder (R, ⊆ for ≤, ⊇ for ≥, ⊂ for <,
      ⊃ for >)
    ∀ e: E, r, r1, r2: R
      e ∈ r == e ∈ set(r);
      -(r1 ∪ r2) == (-r1) ∩ (-r2);
      -(r1 ∩ r2) == (-r1) ∪ (-r2);
      (r1 ∘ r2)⁻¹ == (r2⁻¹) ∘ (r1⁻¹)
    converts
      ∈, ∉, set, dom, range, __⁺, __*, __-__, ×,
      ∪, ∩, ∘, -:R→R, ⁻¹, ⊆, ⊇, ⊂, ⊃,
      domRestrict, image, rangeRestrict

SetToRelation: trait
  % Map a (finitely generated) set
  % to the relation that represents it.
  assumes SetBasics, RelationBasics
  introduces
    relation: C → R
  asserts
    ∀ e: E, s: C
      relation({}) == ⊥;
      relation(insert(e, s)) == [e, e] ∪ relation(s)
  implies
    ∀ e: E, s: C
      e ∈ s == e ⟨ relation(s) ⟩ e
    converts relation
```

The predicates in the next trait are closely related to the theories defined in Section 1.11, but they define the properties of relations treated as values, whereas Section 1.11 defines properties of relations treated as operators. This duplication is a price of not using a higher-order logic in LSL.

```
RelationPredicates: trait
  % Tests for useful properties
  % of individual relations.
  assumes
    RelationBasics,
    RelationOps
  introduces
    antisymmetric, asymmetric, equivalence,
      functional, irreflexive, oneToOne, reflexive,
      symmetric, total, transitive: R → Bool
    into, onto: R, R → Bool
  asserts
    ∀ r, r1, r2: R
      antisymmetric(r) == (r ∩ (r⁻¹)) ⊆ I;
      asymmetric(r) == r ∩ (r⁻¹) = ⊥;
      equivalence(r) ==
        reflexive(r) ∧ symmetric(r) ∧ transitive(r);
      functional(r) == ((r⁻¹) ∘ r) ⊆ I;
      irreflexive(r) == r ∩ I = ⊥;
      oneToOne(r) == r ∘ (r⁻¹) = I;
      reflexive(r) == I ⊆ r;
      symmetric(r) == r = r⁻¹;
      total(r) == dom(r) = I;
      transitive(r) == r = r⁺;
      into(r1, r2) == range(r1) ⊆ set(r2);
      onto(r1, r2) == set(r2) ⊆ range(r1);
  implies converts
    antisymmetric, asymmetric, equivalence,
    functional, irreflexive, oneToOne, reflexive,
    symmetric, total, transitive, into, onto
```

1.9 Graph theory

```
Graph (N, G): trait
  % n1 ⟨ g ⟩ n2 means that there is
  % an edge from n1 to n2 in g
  includes Relation (N for E, G for R)
  introduces
    nodes, undirected: G → G
    isPath: N, N, G → Bool
    stronglyConnected, weaklyConnected: G → Bool
  asserts ∀ n1, n2: N, g: G
    undirected(g) == g ∪ (g⁻¹);
    nodes(g) == dom(g) ∪ range(g);
    isPath(n1, n2, g) == n1 ⟨ g* ⟩ n2;
    stronglyConnected(g) == g* = nodes(g) × nodes(g);
    weaklyConnected(g) ==
      stronglyConnected(undirected(g))
  implies
    ∀ n1, n2: N, g: G
      (stronglyConnected(g) ∧ n1 ∈ nodes(g)
          ∧ n2 ∈ nodes(g))
        ⇒ isPath(n1, n2, g)
```

1.10 Properties of single operators

```
Associative (○, T): trait
  introduces __ ○ __: T, T → T
  asserts ∀ x, y, z: T
    (x ○ y) ○ z == x ○ (y ○ z)

Commutative (○, T, Range): trait
  introduces __ ○ __: T, T → Range
  asserts ∀ x, y: T
    x ○ y == y ○ x

AC (○, T): trait
  introduces __ ○ __: T, T → T
  asserts ∀ x, y, z: T
    (x ○ y) ○ z == x ○ (y ○ z);
    x ○ y == y ○ x
  implies
    Associative,
    Commutative (T for Range)

Idempotent (op, T): trait
  introduces op: T → T
  asserts ∀ x: T
    op(op(x)) == op(x)

Involutive (op, T): trait
  introduces op: T → T
  asserts ∀ x: T
    op(op(x)) == x
```

1.11 Properties of relational operators

Compare with `RelationPredicates`, page 34

```
Antisymmetric (◇): trait
  introduces __ ◇ __: T, T → Bool
  asserts ∀ x, y: T
    (x ◇ y ∧ y ◇ x) ⇒ x = y

Asymmetric (◇): trait
  introduces __ ◇ __: T, T → Bool
  asserts ∀ x, y: T
    x ◇ y ⇒ ¬(y ◇ x)

Functional (◇): trait
  introduces __ ◇ __: T, T → Bool
  asserts ∀ x, y, z: T
    (x ◇ y ∧ x ◇ z) ⇒ y = z;

Irreflexive (◇): trait
  introduces __ ◇ __: T, T → Bool
  asserts ∀ x: T
    ¬(x ◇ x)

OneToOne (◇): trait
  introduces __ ◇ __: T, T → Bool
  asserts ∀ x, y, z: T
    (x ◇ y ∧ x ◇ z) ⇒ y = z;
    (x ◇ z ∧ y ◇ z) ⇒ x = y;

Reflexive (◇): trait
  introduces __ ◇ __: T, T → Bool
  asserts ∀ x: T
    x ◇ x

Symmetric (◇): trait
  introduces __ ◇ __: T, T → Bool
  asserts ∀ x, y: T
    x ◇ y == y ◇ x
  implies Commutative (◇ for ∘, Bool for Range)
```

```
Transitive (◇): trait
  introduces __ ◇ __: T, T → Bool
  asserts ∀ x, y, z: T
    (x ◇ y ∧ y ◇ z) ⇒ x ◇ z

Equivalence: trait
  includes
    (Reflexive, Symmetric, Transitive)(≡ for ◇)

Equality (T): trait
  % This trait is given for documentation only.
  % It is implicit in LSL.
  introduces __ = __, __ ≠ __: T, T → Bool
  asserts
    T partitioned by =
    ∀ x, y, z: T
      x = x;
      x = y == y = x;
      (x = y ∧ y = z) ⇒ x = z;
      x ≠ y == ¬(x = y)
  implies Equivalence (= for ≡)
```

1.12 Orderings

PARTIAL AND TOTAL ORDERS

```
IsPO (≤, T): trait
  % ≤ is a partial order on T
  introduces __≤__: T, T → Bool
  asserts ∀ x, y, z: T
    x ≤ x;
    (x ≤ y ∧ y ≤ z) ⇒ x ≤ z;
    x ≤ y ∧ y ≤ x == x = y
  implies
    Antisymmetric (≤),
    PreOrder,
    Reflexive (≤),
    Transitive (≤)
    T partitioned by ≤


PartialOrder (T): trait
  includes IsPO, DerivedOrders
  implies
    PartialOrder (> for <, < for >,
                  ≥ for ≤, ≤ for ≥),
    StrictPartialOrder (<, T)

IsTO (≤, T): trait
  % ≤ is a total order on T
  introduces __≤__: T, T → Bool
  asserts ∀ x, y, z: T
    x ≤ x;
    (x ≤ y ∧ y ≤ z) ⇒ x ≤ z;
    x ≤ y ∧ y ≤ x == x = y;
    x ≤ y ∨ y ≤ x
  implies IsPO, TotalPreOrder


TotalOrder (T): trait
  includes IsTO, DerivedOrders
  implies
    PartialOrder,
    StrictTotalOrder (<, T),
    TotalOrder (≥ for ≤, ≤ for ≥,
                > for <, < for >)
    T partitioned by <
```

ASSUMPTIONS AND IMPLICATIONS

```
PreOrder (≤, T): trait
  includes Reflexive (≤), Transitive (≤)
  implies ∀ x, y, z: T
    x ≤ x;
    (x ≤ y ∧ y ≤ z) ⇒ x ≤ z
```

```
TotalPreOrder (≤, T): trait
  includes PreOrder
  asserts ∀ x, y: T
    x ≤ y ∨ y ≤ x
```

```
StrictPartialOrder (<, T): trait
  includes Irreflexive (<), Transitive (<)
  implies
    Asymmetric (<)
    ∀ x, y, z: T
      ¬(x < x);
      (x < y ∧ y < z) ⇒ x < z
```

```
StrictTotalOrder (<, T): trait
  includes StrictPartialOrder
  asserts ∀ x, y: T
    x < y ∨ y < x ∨ x = y
```

OPERATOR DEFINITIONS

```
DerivedOrders (T): trait
  % Define any three of the comparison operators,
  % given the fourth
  introduces
    __≤__, __≥__, __<__, __>__: T, T → Bool
  asserts ∀ x, y: T
    x ≤ y == x < y ∨ x = y;
    x < y == x ≤ y ∧ ¬(x = y);
    x ≥ y == y ≤ x;
    x > y == y < x
  implies
    converts ≥, <, >
    converts ≤, <, >
    converts ≤, ≥, >
    converts ≤, ≥, <
```

```
MinMax (T): trait
  assumes TotalOrder
  introduces
    min, max: T, T → T
  asserts ∀ x, y: T
    min(x, y) == if x ≤ y then x else y;
    max(x, y) == if x ≥ y then x else y
  implies
    AC (min, T),
    AC (max, T)
    converts min, max
```

```
LexicographicOrder (E, C): trait
  % "Dictionary" order on C
  assumes
    Container,
    StrictTotalOrder (<, E)
  includes DerivedOrders (C)
  asserts ∀ c1, c2: C
    c1 < c2 ==
        c2 ≠ empty
          ∧ (c1 = empty
              ∨ (if head(c1) = head(c2)
                  then tail(c1) < tail(c2)
                  else head(c1) < head(c2)))
  implies
    TotalOrder (C)
    converts ≤:C,C→Bool, ≥:C,C→Bool,
      <:C,C→Bool, >:C,C→Bool
```

1.13 Lattice theory

```
GreatestLowerBound (T): trait
  introduces
    __ ≤ __: T, T → Bool
    __ ⊓ __: T, T → T
  asserts ∀ x, y, z: T
    (x ⊓ y) ≤ x;
    (x ⊓ y) ≤ y;
    (z ≤ x ∧ z ≤ y) ⇒ z ≤ (x ⊓ y)

Semilattice (T): trait
  assumes PartialOrder
  includes GreatestLowerBound
  introduces
    ⊥: → T
    __⊔ __: T, T → T
  asserts ∀ x, y, z: T
    ⊥ ≤ x;
    x ⊔ y == y ⊔ x;
    x ⊓ y == y ⊓ x;
    x ≤ (x ⊔ y);
    (x ≤ z ∧ y ≤ z) ⇒ (x ⊔ y) ≤ z
  implies
    AbelianMonoid (⊔ for ∘, ⊥ for unit),
    AbelianSemigroup (⊓ for ∘)

Lattice (T): trait
  assumes PartialOrder
  includes Semilattice
  introduces ⊤: → T
  asserts ∀ x: T
    x ≤ ⊤
  implies
    Lattice (⊔ for ⊓, ⊓ for ⊔, ⊤ for ⊥, ⊥ for ⊤,
             ≤ for ≥, ≥ for ≤, < for >, > for <)
```

1.14 Group theory

```
Semigroup: trait
  introduces __∘ __: T, T → T
  asserts ∀ x, y, z: T
    (x ∘ y) ∘ z == x ∘ (y ∘ z)
  implies Associative

LeftIdentity: trait
  introduces
    __ ∘ __: T, T → T
    unit: → T
  asserts ∀ x: T
    unit ∘ x == x

RightIdentity: trait
  introduces
    __ ∘ __: T, T → T
    unit: → T
  asserts ∀ x: T
    x ∘ unit == x

Identity: trait
  includes LeftIdentity, RightIdentity

Monoid: trait
  introduces
    __∘ __: T, T → T
    unit: → T
  asserts ∀ x, y, z: T
    (x ∘ y) ∘ z == x ∘ (y ∘ z);
    unit ∘ x == x;
    x ∘ unit == x
  implies Semigroup, Identity

LeftInverse: trait
  assumes LeftIdentity
  introduces __⁻¹: T → T
  asserts ∀ x: T
    (x⁻¹) ∘ x == unit
```

```
RightInverse: trait
  assumes RightIdentity
  introduces __⁻¹: T → T
  asserts ∀ x: T
    x ∘ (x⁻¹) == unit

Inverse: trait
  assumes Identity, Semigroup
  includes LeftInverse, RightInverse
  implies
    Involutive (__⁻¹ for op)
    ∀ x, y: T
      unit⁻¹ == unit;
      (x ∘ y)⁻¹ == (y⁻¹) ∘ (x⁻¹)

Group: trait
  introduces
    __∘ __: T, T → T
    unit: → T
    __⁻¹: T → T
  asserts ∀ x, y, z: T
    (x ∘ y) ∘ z == x ∘ (y ∘ z);
    unit ∘ x == x;
    (x⁻¹) ∘ x == unit;
  implies Monoid, Inverse

Abelian: trait
  introduces __ ∘ __: T, T → T
  asserts ∀ x, y: T
    x ∘ y == y ∘ x
  implies Commutative (T for Range)

AbelianSemigroup: trait
  includes Abelian, Semigroup
  implies AC

AbelianMonoid: trait
  includes Abelian, Monoid

AbelianGroup: trait
  includes Abelian, Group
```

```
LeftDistributive (+, *, T): trait
  introduces
    __+__, __*__: T, T → T
  asserts ∀ x, y, z: T
    x * (y + z) == (x * y) + (x * z)

RightDistributive (+, *, T): trait
  introduces
    __+__, __*__: T, T → T
  asserts ∀ x, y, z: T
    (y + z) * x == (y * x) + (z * x)

Distributive (+, *, T): trait
  includes LeftDistributive, RightDistributive

Ring: trait
  includes
    AbelianGroup (+ for ∘, 0 for unit, -__ for ⁻¹),
    Semigroup (* for ∘),
    Distributive (+, *, T)

RingWithUnit: trait
  includes Ring, Monoid (* for ∘, 1 for unit)

Field: trait
  includes
    RingWithUnit,
    Abelian (* for ∘)
  introduces __⁻¹: T → T
  asserts ∀ x: T
    x ≠ 0 ⇒ x * (x⁻¹) = 1
```

1.15 Number theory

This section presents a series of traits dealing with operators on whole numbers. The following section deals with operators on rational and floating point numbers.

DATA TYPES

```
Natural (N): trait
  % The usual operators on the natural numbers,
  % starting from 0.
  includes
    ArithOps (N),
    DecimalLiterals,
    Exponentiation (N),
    MinMax (N),
    TotalOrder (N)
  introduces
    __ ⊖ __: N, N → N
  asserts
    N generated by 0, succ
    ∀ x, y: N
      succ(x) ≠ 0;
      succ(x) = succ(y) == x = y;
      x < succ(x);
      0 ⊖ x == 0;
      x ⊖ 0 == x;
      succ(x) ⊖ succ(y) == x ⊖ y
  implies
    NaturalOrder
    N generated by 0, 1, +
    ∀ x, y: N
      x ⊖ x == 0;
      x ≤ y == x ⊖ y = 0
    converts 1:→N, +, ⊖, *, div, mod,
        **, min, max, ≤, ≥, <, >
      exempting ∀ x: N
        div(x, 0), mod(x, 0)
```

```
Positive (P): trait
  % Basic operators on natural numbers,
  % starting from 1
  includes DecimalLiterals (P for N), TotalOrder (P)
  introduces
    1: → P
    succ: P → P
    __+__, __*__: P, P → P
  asserts
    P generated by 1, succ
    ∀ x, y: P
      x + 1 == succ(x);
      x + succ(y) == succ(x + y);
      x*1 == x;
      x*succ(y) == x + (x*y);
      x < succ(x)
  implies
    NaturalOrder (P for N, 1 for 0)
    P generated by 1, +
    converts +, *, ≤, ≥, <, >
```

```
IntCycle (first, last, N): trait
  % A finite subrange of the integers that includes 0,
  % and wraps at succ(last)
  includes
    ArithOps (N),
    DecimalLiterals,
    MinMax (N),
    TotalOrder (N)
  introduces
    first, last: → N
    pred, -__, abs: N → N
    __-__: N, N → N
  asserts
    N generated by 0, succ
    ∀ x, y: N
      succ(last) == first;
      pred(succ(x)) == x;
      succ(pred(x)) == x;
      -0 == 0;
      -succ(x) == pred(-x);
      abs(x) == if x < 0 then -x else x;
      x - y == x + (-y);
      x ≠ last ⇒ x < succ(x)
  implies
    Distributive (+, *, N),
    RingWithUnit (N for T)
    N generated by 0, pred
    ∀ x: N
      pred(first) == last;
      first ≤ x;
      x ≤ last;
      -(-x) == x
    converts
      pred, -__:N→N, abs, __-__:N,N→N,
      1:→N, +, *, max, min, ≤, ≥, <, >

SignedInt (maxSigned, N): trait
  % Typical machine arithmetic, signed complement.
  includes IntCycle (minSigned, maxSigned, N)
  asserts equations
    succ(minSigned) == -maxSigned
  implies equations
    minSigned + maxSigned == -1;
    abs(minSigned) == minSigned
```

```
UnsignedInt (maxUnsigned, N): trait
  % Typical machine arithmetic, unsigned.
  includes IntCycle (0, maxUnsigned, N)
```

ASSUMPTIONS AND IMPLICATIONS

`Enumerable` requires only that each value of sort `N` must be reachable by applying `succ` to `0` a finite number of times. `Infinite` requires that the values yielded by `succ` are all distinct. The inclusion of `TotalOrder` in `NaturalOrder` ensures that `succ(x)` is always greater than `x`, and hence that there are infinitely many distinct values of sort `N`.

```
Enumerable (N): trait
  introduces
    0: → N
    succ: N → N
  asserts
    N generated by 0, succ

Infinite (N): trait
  introduces
    0: → N
    succ: N → N
  asserts ∀ x, y: N
    succ(x) ≠ 0;
    succ(x) = succ(y) == x = y

NaturalOrder (N): trait
  % The natural numbers with an ordering
  includes
    Enumerable (N),
    TotalOrder (N)
  asserts ∀ x: N
    x < succ(x)
  implies
    Infinite (N)
    ∀ x, y: N
      0 ≤ x;
      x < succ(y) == x ≤ y;
      succ(x) < succ(y) == x < y
    converts ≤, ≥, <, >
```

OPERATOR DEFINITIONS

```
Addition (N): trait
  % Define the operator + in terms of 0 and succ
  includes AbelianMonoid(+ for ∘, 0 for unit, N for T)
  introduces
    0: → N
    succ: N → N
    __+__: N, N → N
  asserts ∀ x, y: N
    x + 0 == x;
    x + succ(y) == succ(x + y)

Multiplication (N): trait
  % Define the operator * in terms of 0, succ, and +
  includes
    AbelianMonoid (* for ∘, 1 for unit, N for T),
    Addition (N)
  introduces
    1: → N
    __*__: N, N → N
  asserts ∀ x, y: N
    1 == succ(0);
    x * 0 == 0;
    x * succ(y) == x + (x * y)

ArithOps (N): trait
  % Defines operators div and mod relative to + and *
  % for positive denominators
  assumes TotalOrder (N)
  includes Multiplication (N)
  introduces
    div, mod: N, N → N
  asserts ∀ x, y: N
    y > 0
      ⇒ (0 ≤ mod(x, y)
          ∧ mod(x, y) < y
          ∧ (mod(x, y) + (div(x, y) * y)) = x)
```

```
Exponentiation (T): trait
  % Repeatedly apply an infix * operator
  assumes
    Enumerable (N),
    Monoid (* for ∘, 1 for unit)
  introduces __**__: T, N → T
  asserts ∀ x: T, y: N
    x**0 == 1;
    x**succ(y) == x * (x**y)
  implies ∀ x: T
    x**succ(0) == x

IntegerAndNatural (Int, N): trait
  % Conversions between Int's and N's
  includes
    Integer (Int),
    Natural (N)
  introduces
    int: N → Int
    nat: Int → N
  asserts ∀ n: N
    int(0) == 0;
    int(succ(n)) == succ(int(n));
    nat(int(n)) == n

IntegerAndPositive (Int, P): trait
  % Conversions between Int's and P's
  includes
    Integer (Int),
    Positive (P)
  introduces
    int: P → Int
    pos: Int → P
  asserts ∀ p: P
    int(1) == 1;
    int(succ(p)) == succ(int(p));
    pos(int(p)) == p
```

1.16 Floating point arithmetic

The trait `Rational` provides enough of a theory of rational arithmetic to specify the properties of floating point arithmetic.

```
Rational: trait
  % For use in the trait FloatingPoint.
  includes
    Exponentiation (Q for T, P for N),
    IntegerAndPositive (Int, P),
    MinMax (Q),
    TotalOrder (Q)
  introduces
    __/__: Int, P → Q
    0, 1: → Q
    -__, __⁻¹, abs: Q → Q
    __+__, __*__, __-__, __/__: Q, Q → Q
  asserts
    Q generated by __/__:Int,P→Q
    ∀ i, i1, i2: Int, p, p1, p2, p3: P, q, q1, q2: Q
      0/p == 0;
      int(p)/p == 1;
      i1/p1 = i2/p2 == i1 * int(p2) = i2 * int(p1);
      -(i/p) == (-i)/p;
      (int(p1)/p2)⁻¹ == int(p2)/p1;
      (-q)⁻¹ == -(q⁻¹);
      abs(i/p) == abs(i)/p;
      (i1/p) + (i2/p) == (i1 + i2)/p;
      (i1/p1) * (i2/p2) == (i1 * i2)/(p1 * p2);
      q1 - q2 == q1 + (-q2);
      q1/q2 == q1 * (q2⁻¹);
      (i1/p) < (i2/p) == i1 < i2
  implies
    AC (+, Q),
    AC (*, Q),
    Field (Q for T)
    ∀ i, i1, i2: Int, p, p1, p2, p3: P, q: Q
      q + 0 == q;
      -q == 0 - q;
      (i1/p) - (i2/p) == (i1 - i2)/p;
      q * 0 == 0;
      q * 1 == q;
      q⁻¹ == 1/q;
      (i/p1)/(int(p2)/p3) == (i * int(p3))/(p1 * p2)
  converts
```

```
    0:→Q, 1:→Q, -:Q →Q, ⁻¹, abs:Q →Q,
    +:Q,Q→Q, -:Q,Q→Q, *:Q,Q→Q, /:Q,Q→Q,
    **:Q,P→Q, min:Q,Q→Q, max:Q,Q→Q,
    <:Q,Q→Bool, >:Q,Q→Bool,
    ≤:Q,Q→Bool, ≥:Q,Q→Bool
  exempting 0⁻¹
```

The following traits define a theory of floating point arithmetic that is weak enough to be satisfied by many floating point implementations, yet strong enough to allow reasoning about floating point arithmetic. Careful analysis of any particular floating point system should lead to tighter bounds on the errors due to inexact arithmetic, and might even lead to some useful identities, such as $(f_1 + f_2) + f_3 = f_1 + (f_2 + f_3)$.

The basic idea is this: Every floating point number exactly represents some rational number, returned by the operator `rational`. Each floating point operator approximates a corresponding rational operator, but cannot always be be exact. The exact answer may not even be representable. Furthermore, floating point arithmetic does not generally guarantee to produce even the closest representable value. So each floating point operator may introduce an error that depends on:

- the magnitude of the operand(s),
- the magnitude of the exact and approximate results,
- properties of the floating point representation used.

Three parameters characterize the representation: `smallest` and `largest` denote the least and the greatest representable positive values, respectively, and `gap`, the largest relative difference between any pair of consecutive representable positive values. `FPAssumptions` specifies relations that must hold among these parameters and the operator `rational` (which converts floating point numbers to their exact rational values) in order for `FloatingPoint` to characterize a valid floating point number system.

```
FPAssumptions (smallest, largest,
               gap, rational): trait
  includes Rational
  introduces
    smallest, largest, gap: → Q
    rational: F → Q
    float: Q → F
    0, 1: → F
  asserts ∀ f: F
    smallest > 0;
    largest > smallest;
    rational(0) == 0;
    rational(1) == 1;
    rational(f) ≠ 0 ⇒ abs(rational(f)) ≥ smallest;
    rational(f) ≤ largest;
    gap > 0;
    float(rational(f)) == f;
```

The predicate `approx(f, q, t)` compares the result `f` of a floating point operation to the exact rational value `q` of that operation; the predicate is true if the result is "close enough" to the exact value (i.e., within a tolerance `t`), or if the exact value is too big to be represented.

We have not axiomatized the properties of the IEEE standard's non-numeric floating point values (NaN's). We leave that as an exercise for numerical analysts, in the expectation that an accurate characterization is separable from the numerical properties. It might be more complex than anything we have specified in this handbook.

```
FloatingPoint (smallest, largest,
               gap, rational): trait
  assumes FPAssumptions
  includes
    Rational,
    TotalOrder (F)
  introduces
    mag: F → Q
    approx: F, Q, Q → Bool
    -__, abs, __⁻¹: F → F
    __+__, __*__, __-__, __/__: F, F → F
  asserts
    F generated by float
    ∀ f, f1, f2: F, q, t: Q
      f1 ≤ f2 == rational(f1) ≤ rational(f2);
      mag(f) == abs(rational(f));
      approx(f, q, t) ==
        abs(q) ≤ largest
          ⇒ abs(rational(f) - q)
              ≤ (smallest +
                  (gap*(mag(f) + abs(q) + t)));
      approx(-f, -rational(f), 0);
      f ≠ 0 ⇒ approx(f⁻¹, rational(f)⁻¹, 0);
      approx(abs(f), mag(f), 0);
      approx(f1 + f2, rational(f1) + rational(f2),
             mag(f1) + mag(f2));
      approx(f1 * f2, rational(f1) * rational(f2), 0);
      approx(f1 - f2, rational(f1) - rational(f2),
             mag(f1) + mag(f2));
      f2 ≠ 0
        ⇒ approx(f1/f2, rational(f1)/rational(f2), 0)
```