Tecton Description of STL Container and Iterator Concepts

Rüdiger Loos
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
loos@informatik.uni-tuebingen.de

David Musser
Computer Science Department
Rensselaer Polytechnic Institute
musser@cs.rpi.edu

Draft, November 20, 2001

Abstract

The various concepts of containers and iterators used in the C++ Standard Template Library (STL) are formally described in this paper using the Tecton concept description language.

Contents

1 Introduction

We present formal specifications of the C++ Standard Template Library (STL) [10, 8, 9] container and iterator concepts, expressing them in the Tecton concept description language [4, 3, 7]. We have subjected these specifications to syntactic-, type-, and limited semantic checking as currently supported by the ESTALD translator [?]. In fact, successive iterations on this specification have served as some of the most stringent tests of ESTALD to date.

Our starting point was the set of semi-formal concept descriptions given in [9], since that is already organized as a concept hierarchy and is more complete in its statement of semantics than the actual C++ standard [2]. There are some places noted where we have to depart from [9]. In some cases the differences are merely notational, to meet technical requirements of Tecton, but in others there appear to be subtle inconsistencies in the specification in [9] (which also appear in the ANSI/ISO C++ standard). In general, we believe attempting to write formal specifications is an effective means of detecting errors that can escape more commonplace methods, and this exercise bears that observation out.

We present the specification in the "literate programming" style advocated by Knuth [5], adapted here for specification rather than program development purposes. This style allows us to present the specification in "parts" in a top-down, bottom-up, or mixed order, with the proper order as required by Tecton being composed by the literate program extraction tool. For the latter, we use NUWEB [1].

2 Overview of Concept Definitions and Lemmas

The overall specification is written to a file, `container.tec`, for submission to ESTALD. We show here the names and order of the parts that are developed in subsequent sections.

`"container.tec"` 1 ≡
⟨Basic concepts 3a⟩
⟨Object concepts 3b⟩
⟨Equality-comparable concept 4b⟩
⟨Strict weak order concept 7d⟩
⟨Less-than-comparable concept 7e⟩
⟨Basic-container concept 4c⟩
⟨Trivial iterator concept 4d⟩
⟨Input iterator concept 5a⟩
⟨Output iterator concept 5b⟩
⟨Container concept 6a⟩
⟨Forward iterator concept 6b⟩
⟨Extension of forward iterator concept 6c⟩
⟨Constness extensions of forward iterator concept 7a⟩
⟨Transitivity of validity of ranges 7b⟩

⟨Forward container concept 7c⟩

⟨Range equivalence concept 8b⟩

⟨Access equivalence concept 9a⟩

⟨Lexicographical order concept 7f⟩

⟨Forward-container-with-less-than 8a⟩

⟨Indexing concepts 9b⟩

⟨Basic sequence concept 9c⟩

⟨Sequence concept 11a⟩

⟨Default construction lemma 10c⟩

⟨Front insertion sequence concept 12b⟩

⟨Back insertion sequence concept 12c⟩

⟨Bidirectional iterator concept 12d⟩

⟨Random-access iterator concept 13a⟩

⟨Reversible container concept 13b⟩

⟨Some invariants of reversible containers 13c⟩

⟨Random access container concept 14a⟩

⟨Vector concept 14b⟩

⟨Deque concept 14c⟩

⟨Spliceable container concept 15a⟩

⟨List concept 15b⟩

⟨Associative container concept 18a⟩

⟨Sorted associative container concept 18b⟩

⟨Hashed associative container concept 19a⟩

⟨Multiple associative container concept 19b⟩

⟨Unique associative container concept 19c⟩

⟨Simple associative container concept 19d⟩

⟨Pair and associative-pair concepts 20a⟩

⟨Pair associative container concept 20b⟩

⟨Combinations of associative container concepts 20c⟩

3 Basic Concepts

A long term goal is to develop an extensive library of Tecton concepts that new specification efforts will be able to build upon rather than working from scratch. The logical and physical organization of the library is still to be worked out, but we are currently making limited attempts to reuse previously developed concept descriptions. Here we import, via the part named `Through Natural`, some basic algebraic concepts as developed in [6]; for the reader's convenience this part is listed, without discussion, in an appendix of the present paper.

⟨Basic concepts 3a⟩ ≡

⟨Through Natural 22⟩

```
Definition: Function-set
refines Domain [with functions as domain];
uses Domain, Range;
introduces value(functions, domain) -> range.

Definition: Binary-function-set
refines Domain [with functions as domain];
uses Domain [with domain1 as domain], Domain [with domain2 as domain], Range;
introduces value(functions, domain1, domain2) -> range.

Abbreviation: Predicate-set
is Function-set [with predicates as functions, bool as range].

Abbreviation: Binary-predicate-set
is Binary-function-set [with binary-predicates as functions, bool as range].
```

Used in part 1.

⟨Object concepts 3b⟩ ≡

⟨Object, accessible object, and writable object concepts 3c⟩

⟨Assignable object concept 3d⟩

⟨Default-constructible object concept 4a⟩

Used in part 1.

⟨Object, accessible object, and writable object concepts 3c⟩ ≡

```
Abbreviation: Object is Domain [with objects as domain].

Definition: Element
  refines Domain [with elements as domain];
  introduces
    construct-element -> elements.

Definition: Accessible refines Object;
uses Element;
introduces access(objects) -> elements.

Precedence: nonassociative{=, :=}.

Definition: Writable refines Object;
uses Element;
introduces :=(objects, elements) -> objects,
  constant(objects) -> bool.
```

Used in part 3b.

⟨Assignable object concept 3d⟩ ≡

```
Definition: Assignable refines Accessible, Writable;
introduces
   copy(objects) -> objects,
   swap(objects, objects) -> (objects, objects);
requires (for x, x1, y, y1: objects; z: objects; e: elements)
   access(x := e) = e,
   access(copy(z)) = access(z),
//   swap(x, y) = (y, x).
```

```
swap(x, y; x1, y1) where x1 = y, y1 = x.
```

Used in part 3b.

⟨Default-constructible object concept 4a⟩ ≡

```
Definition: Default-constructible refines Accessible;
introduces construct -> objects.
```

Used in part 3b.

⟨Equality-comparable concept 4b⟩ ≡

```
Definition: Equality-comparable
  refines Object;
  introduces  =(objects, objects) -> bool,
             !=(objects, objects) -> bool.
```

Used in part 1.

4 Container and Iterator Concepts

Initially we define a basic container concept very simply, but after defining the input iterator concept we define the container concept with operations that depend on iterators.

⟨Basic-container concept 4c⟩ ≡

```
Definition: Basic-container
refines Assignable [with containers as objects];
uses Accessible.
```

Used in part 1.

We depart here from [9], both in first defining a Basic-container concept before defining Container, and in requiring the objects in the container to be only Accessible rather than Assignable. The first difference is due only to technical requirements of Tecton, but the second is necessary because [9] appears to be in error in requiring Assignable for the objects. The error becomes apparent when the Associative Container concept is introduced in [9] as a refinement of Container but with the statement that the objects in an associative container are not necessarily assignable. (For containers other than associative containers, such as Forward-container, Sequence, etc., we obtain the assignable property for contained objects as a combination of the accessible and writable properties as introduced via the Forward-iterator concept below.)

Although it is a unary operator on C++ pointers and on their STL generalization, iterators, the dereferencing operator, `*`, must be expressed here as a binary operator depending on the container as well as the iterator.

⟨Trivial iterator concept 4d⟩ ≡

```
Definition: Trivial-iterator
refines Assignable [with iterators as objects],
        Default-constructible [with iterators as objects],
        Equality-comparable [with iterators as objects];
uses Accessible, Basic-container;
introduces *(containers, iterators) -> objects.
```

Used in part 1.

Note that the `objects` sort used as the range of `*` is from the Accessible concept. So trivial iterators give a means of obtaining accessible, but not necessarily assignable, objects.

In defining input iterators the main difficulty is to avoid making overly strong assertions about the meaning of `++`, since we want to allow models in which not only the iterator but also the container into which it points is changed by the operation. Thus we make the range of `++` the cartesian product of `containers` and `iterators` rather than just `iterators`. As with the dereferencing operator, `++` is a unary operator on C++ pointers and on their STL generalization, iterators, but we must express it here as a binary operator depending on the container as well as the iterator.

⟨Input iterator concept 5a⟩ ≡

```
Precedence: nonassociative{=} < nonassociative{++, --} < {+}.

Definition: Input-iterator refines Trivial-iterator;
uses Natural;
introduces ranges,
  ++(containers, iterators) -> (containers, iterators),
  range(containers, iterators, iterators) -> ranges,
  in(iterators, ranges) -> bool,
  next(containers, iterators, naturals) -> iterators (private),
  valid(ranges) -> bool,
  size(ranges) -> naturals;
requires (for i, i1, j, p, q: iterators; c, c1, d: containers; n: naturals)
  next(c, i, 0) = i,
  next(c, i, n + 1) = next(d, p, n) where c++i = (d, p),
  valid(range(c, i, j)) = ((for some k: naturals) next(c, i, k) = j),
  valid(range(c, i, j)) and valid(range(c, p, q)) implies
   (range(c, i, j) = range(c, p, q)) =
     if i = j then p = q
       else i = p
            and range(c1, i1, j) = range(c1, i1, q)
            where c++i = (c1, i1),
  valid(range(c, i, j)) implies
    (p in range(c, i, j)) = (i != j and (p = i or p in range(c1, i1, j))
                             where c++i = (c1, i1)),
  valid(range(c, i, j)) implies
    size(range(c, i, j)) = if i = j then 0
                           else size(range(d, p, j)) where c++i = (d, p).
```

Used in part 1.

The output iterator concept gives us the ability to assign to the objects to which the iterators point, but not necessarily to access them.

⟨Output iterator concept 5b⟩ ≡

```
Definition: Output-iterator
refines Assignable [with iterators as objects],
        Default-constructible [with iterators as objects];
uses Writable, Basic-container;
introduces
  *(containers, iterators) -> objects,
  ++(containers, iterators) -> (containers, iterators).
```

Used in part 1.

We can now define the container concept:

⟨Container concept 6a⟩ ≡

```
Definition: Container
refines Basic-container;
uses Input-iterator;
introduces
  nonempty-containers < containers,
  begin(containers) -> iterators,
  end(containers) -> iterators,
  size(containers) -> naturals,
  empty(containers) -> bool;
requires (for x: containers)
  x: nonempty-containers = not empty(x),
  size(x) = size(range(x, begin(x), end(x))),
  empty(x) = (size(x) = 0),
  valid(range(x, begin(x), end(x))).
```

Used in part 1.

5 Forward Iterator and Forward Container Concepts

In the forward iterator concept the ++ function is required to leave the container unchanged.

⟨Forward iterator concept 6b⟩ ≡

```
Definition: Forward-iterator
refines Input-iterator, Output-iterator;
introduces ++(containers, iterators) -> iterators;
requires (for i, p: iterators; c, c1: containers)
  c++i = (c1, p) implies c1 = c,
  c++i = p where c++i = (c, p).
```

Used in part 1.

The above formulation depends on being able to overload function identifiers and resolve them where there is no difference in the argument types, only in the range type. [[If this assumption causes too much trouble, we could avoid it by using some other identifier in place of ++ in the input iterator and output iterator concepts.]]

Note that Input-iterator uses Accessible and Output-iterator uses Writable (in both cases giving properties to the objects to which the iterators point), and both of these concepts together imply Assignable. Since Forward-iterator refines both Input-iterator and Output-iterator, we may treat Assignable as a formal parameter of Forward-iterator. This fact may be stated explicitly in Tecton as an extension:

⟨Extension of forward iterator concept 6c⟩ ≡

```
Extension: Forward-iterator uses Assignable.
```

Used in part 1.

The legality of this extension depends on the above mentioned relationships.

We also extend Forward-iterator to have some predicates relating to "constness." These predicates are not fully defined here, but they do obey a constraint among themselves.

⟨Constness extensions of forward iterator concept 7a⟩ ≡

```
Extension: Forward-iterator uses Assignable.

Extension: Forward-iterator
introduces
  constant(containers) -> bool,
  constant-in-object-type(containers) -> bool,
  points-to-constant(iterators) -> bool;
requires (for c: containers; i: iterators)
  constant(c*i) = (constant-in-object-type(c) or constant(c)
                   or points-to-constant(i)).
```

Used in part 1.

These predicates can be used to model C++ declarations involving the `const` attribute and `const_iterators`.

Another observation is that with forward iterators the validity of ranges is transitive.

⟨Transitivity of validity of ranges 7b⟩ ≡

```
Lemma: Forward-iterator
obeys (for i, j, k: iterators; c: containers)
  valid(range(c, i, j)) and valid(range(c, j, k))
    implies valid(range(c, i, k)).
```

Used in part 1.

A transitivity property could be stated for even input iterators, but it would be more complicated (and weaker) because the container can be modified by `++`.

The forward container concept is now defined in terms of the container concept by merely substituting the forward iterator concept for that of input iterators.

⟨Forward container concept 7c⟩ ≡

```
Definition: Forward-container
refines Container [with Forward-iterator as Input-iterator].
```

Used in part 1.

When there is an ordering requirement on the object concept, we have an ordering on forward containers. Originally this requirement was stated as a strict total order, but the weaker notion of *strict weak order* suffices.

⟨Strict weak order concept 7d⟩ ≡

```
Definition: Strict-weak-order
refines Strict-partial-order, Transitive [with equivalent as R];
requires (for x, y: domain)
    equivalent(x, y) = (not(x < y) and not(y < x)).
```

Used in part 1.

⟨Less-than-comparable concept 7e⟩ ≡

```
Abbreviation: Less-than-comparable
  is Strict-weak-order.
```

Used in part 1.

⟨Lexicographical order concept 7f⟩ ≡

```
Definition: Lexicographical-order
uses Forward-iterator,
     Strict-weak-order [with Object as Domain, objects as domain];
introduces
  less-than(containers, containers,
```

```
                iterators, iterators, iterators, iterators) -> bool;
requires (for c, d: containers; i, j, k, l: iterators)
  less-than(c, d, i, i, k, k) = false,
  k != l implies less-than(c, d, i, i, k, l),
  valid(range(c, i, j)) and i != j implies
     less-than(c, d, i, j, k, k) = false,
  valid(range(c, i, j)) and i != j and
  valid(range(d, k, l)) and k != l implies
       less-than(c, d, i, j, k, l) =
        (c*i < d*k) or
         c*i = d*k and less-than(c, d, c++i, j, d++k, l).
```

Used in part 1.

⟨Forward-container-with-less-than 8a⟩ ≡

```
Definition: Forward-container-with-less-than
refines Forward-container [with Strict-weak-order as Equality-comparable];
uses Lexicographical-order;
introduces <(containers, containers) -> bool;
requires (for x, y: containers)
  (x < y) = less-than(x, y, begin(x), end(x), begin(y), end(y)).
```

Used in part 1.

If objects of a container have an equivalence relation, `equivalent-to`, defined on them, we define equivalence of ranges of iterators as follows:

⟨Range equivalence concept 8b⟩ ≡

```
Precedence: nonassociative{=, equivalent-to, access-equivalent-to}
            < prefix{P}.

Definition: Range-equivalence
uses Forward-iterator,
     Equivalence-relation [with Object as Domain, objects as domain,
                                equivalent-to as R];
introduces
  equivalent-range(containers, containers, iterators, iterators, iterators)
     -> bool;
requires (for c, d: containers; i, j, k: iterators)
  equivalent-range(c, d, i, i, k),
  valid(range(c, i, j)) and i != j implies
     equivalent-range(c, d, i, j, k) =
       (c*i equivalent-to d*k
        and equivalent-range(c, d, c++i, j, d++k)).

Definition: Forward-container-with-equivalence
refines Forward-container;
uses Range-equivalence;
introduces equivalent-to(containers, containers) -> bool;
requires (for x, y: containers)
  (x equivalent-to y) =
   (size(x) = size(y) and
      equivalent-range(x, y, begin(x), end(x), begin(y))).
```

Used in part 1.

A fundamental kind of equivalence that is used with the above definition is access equivalence:

⟨Access equivalence concept 9a⟩ ≡

```
Definition: Access-equivalence
refines Equivalence-relation [with Accessible as Domain, objects as domain,
                              access-equivalent-to as R];
requires (for x, y: objects)
  (x access-equivalent-to y) = (access(x) = access(y)).
```

Used in part 1.

6 Sequence Concept

In the sequence concept we strengthen the container requirements sufficiently to guarantee the elements are maintained in a fixed order, obtainable (repeatably) by iterating through the container. Constructors and insertion and deletion functions are provided, and their use determines the order in which the elements appear. In order to specify the order precisely, we first introduce several indexing-related concepts.

⟨Indexing concepts 9b⟩ ≡

```
Abbreviation: Index-set is Domain [with indices as domain].

Definition: Indexed-container
refines Container;
uses Index-set;
introduces access(containers, indices) -> elements.

Definition: Least-in-partial-order refines Partial-order;
introduces least -> domain;
requires (for x: domain) least <= x.

Definition: Enumerative-total-order
refines Least-in-partial-order, Total-order;
uses Natural;
introduces ranges,
     next(domain) -> domain,
     +(domain, naturals) -> domain,
     -(domain, domain) -> naturals,
     range(domain, domain) -> ranges,
     in(domain, ranges) -> bool;
generates domain freely using least, next;
requires (for x, y, z: domain; n: naturals)
     x < next(x),
     x + 0 = x,
     x + next(n) = next(x + n),
     y = x + n implies y - x = n,
     x in range(y, y) = false,
     y < z implies (x in range(y, z)) = (x = y or (x in range(next(y), z))).
```

Used in part 1.

The sequence concept introduces several new functions, including two versions each of insertion and deletion, which have relatively complex semantics. Here we first introduce a basic sequence concept without insertion operations, so that it can also be used in the definition of associative containers (which have a different notion of insertion from that of sequences).

⟨Basic sequence concept 9c⟩ ≡

```
Definition: Basic-sequence
refines
  Forward-container-with-equivalence
     [with Access-equivalence as Equivalence-relation,
```

```
            access-equivalent-to as equivalent-to,
            sequences as containers],
    Default-constructible [with sequences as objects],
    Indexed-container [with Enumerative-total-order as Index-set,
                            indices as domain,
                            sequences as containers,
                            nonempty-sequences as nonempty-containers];
  introduces
    position(sequences, iterators) -> indices,
    construct(sequences, iterators, iterators) -> sequences,
    erase(nonempty-sequences, iterators) -> sequences,
    erase(sequences, iterators, iterators) -> sequences;
  requires (for c, c1: sequences; d: nonempty-sequences;
                i, i1, j, k: iterators; e: elements;
                p, q: indices; n: naturals)
```

⟨Axioms for position function 10a⟩ ,

⟨Axioms for construct functions 10b⟩ ,

⟨Axioms for erase functions 10d⟩ .

Used in part 1.

⟨Axioms for position function 10a⟩ ≡

```
  position(c, begin(c)) = least,
  position(c, i) = p and p < least + size(c)
    implies position(c, c++i) = next(p),
  position(c, i) < least + size(c) implies
      access(c*i) = access(c, position(c, i))
```

Used in part 9c.

The `access` function on the left of the above equation is from the Accessible concept, while the one on the right is from the Indexed-container concept. Note that this axiom is written using `d`, an identifier declared to be of the `nonempty-sequence` sort.

⟨Axioms for construct functions 10b⟩ ≡

```
  end(construct) = begin(construct)
```

Used in part 9c.

As a trivial consequence of the axiom about `construct` and the `size` axioms we have the following

⟨Default construction lemma 10c⟩ ≡

```
  Lemma: Basic-sequence obeys size(construct) = 0.
```

Used in part 1.

⟨Axioms for erase functions 10d⟩ ≡

```
  i in range(d, begin(d), end(d)) and c1 = erase(d, i) implies
     size(c1) = size(d) - 1 and
     ((for p, q: indices)
        (p in range(least, position(d, i))
           implies access(c1, p) = access(d, p)) and
        (q in range(position(d, i), least + size(d))
           implies access(c1, q) = access(d, next(q)))),
  i in range(d, begin(d), end(d)) and j in range(d, begin(d), end(d))
    and position(d, i) < position(d, j) and size(d) >= size(range(d, i, j))
    and c1 = erase(d, i, j)
     implies
```

```
        size(c1) = size(d) - size(range(d, i, j))
        and ((for p, q: indices)
             (p in range(least, position(d, i))
                implies access(c1, p) = access(d, p))
             and (q in range(position(d, i), least + size(d))
               implies access(c1, q) = access(d, q + size(range(d, i, j)))))
```

Used in part 9c.

The sequence concept is a refinement of the basic sequence concept with functions for inserting elements and getting the front element.

⟨Sequence concept 11a⟩ ≡

```
Definition: Sequence
refines Basic-sequence;
introduces
  construct(naturals, elements) -> nonempty-sequences,
  construct(naturals) -> nonempty-sequences,
  insert(sequences, iterators, elements) -> (sequences, iterators),
  insert(sequences, iterators, sequences, iterators, iterators) -> sequences,
  front(sequences) -> objects;
requires (for c, c1, c2: sequences; d: nonempty-sequences;
          i, i1, j, k: iterators; e: elements; p, q: indices; n: naturals)
  front(c) = c*begin(d),
  ⟨Axioms for construct 11b⟩ ,
  ⟨Axiom for inserting one element 11c⟩ ,
  ⟨Axiom for inserting a range of elements 12a⟩ .
```

Used in part 1.

⟨Axioms for construct 11b⟩ ≡

```
c1 = construct(c, j, k) implies
   size(c1) = size(range(c, j, k)) and
    ((for p: indices) p in range(least, least + size(c1))
         implies access(c1, p) = access(c, position(c, j) + (p - least))),
c1 = construct(n, e) implies
   size(c1) = n and
    ((for p: indices) p in range(least, least + n)
        implies access(c1, p) = e),
construct(n) = construct(n, construct-element)
```

Used in part 11a.

⟨Axiom for inserting one element 11c⟩ ≡

```
insert(c, i, e) = (d, i1) implies
   size(d) = size(c) + 1 and access(d*i1) = e
   and position(d, i1) = position(c, i)
   and ((for p: indices)
         (p in range(least, position(c, i))
             implies access(d, p) = access(c, p))
         and (p in range(position(c, i), least + size(c))
             implies access(d, next(p)) = access(c, p)))
```

Used in part 11a.

⟨Axiom for inserting a range of elements 12a⟩ ≡

```
c2 = insert(c, i, c1, j, k) implies
   size(c2) = size(c) + size(range(c1, j, k))
   and ((for p: indices)
         (p in range(position(c, i), position(c, i) + size(range(c1, i, j)))
          implies access(c2, p) =
                    access(c1, position(c, j) + (p - position(c, i))))
        and
         (p in range(least, position(c, i))
             implies access(c2, p) = access(c, p))
        and
         (p in range(position(c, i), least + size(c))
             implies access(c1, p + size(range(c1, j, k))) = access(c, p)))
```

Used in part 11a.

7 Front Insertion Sequence Concept

⟨Front insertion sequence concept 12b⟩ ≡

```
Definition: Front-insertion-sequence
refines Sequence;
introduces
  push-front(sequences, objects) -> nonempty-sequences,
  pop-front(nonempty-sequences) -> sequences;
requires (for s, s1: sequences; x: objects)
  access(front(push-front(s, x))) = access(x),
  pop-front(push-front(s, x)) access-equivalent-to s.
```

Used in part 1.

8 Back Insertion Sequence Concept

⟨Back insertion sequence concept 12c⟩ ≡

```
Definition: Back-insertion-sequence refines Sequence;
introduces
  back(nonempty-sequences) -> objects,
  push-back(sequences, objects) -> nonempty-sequences,
  pop-back(nonempty-sequences) -> sequences;
requires (for s: sequences; s1: nonempty-sequences; x: objects; i: iterators)
  back(s1) = s1*i where s1++i = end(s1),
  access(back(push-back(s, x))) = access(x),
  pop-back(push-back(s, x)) access-equivalent-to s.
```

Used in part 1.

⟨Bidirectional iterator concept 12d⟩ ≡

```
Definition: Bidirectional-iterator refines Forward-iterator;
introduces --(containers, iterators) -> iterators;
requires (for c: containers; i, j: iterators)
  c++j = i implies c -- i = j.
```

Used in part 1.

9 Random Access Iterator Concept

⟨Random-access iterator concept 13a⟩ ≡

```
Definition: Random-access-iterator
refines Bidirectional-iterator,
  Less-than-comparable [with iterators as domain];
uses Container;
introduces
// +(.. and -(.. with 3 args currently not accepted
  plus(containers, iterators, naturals) -> iterators,
  minus(containers, iterators, naturals) -> iterators,
  brackets(containers, iterators, naturals) -> objects;
requires (for c: containers; i, j: iterators; n: naturals)
  i in range(c, begin(c), end(c)) and size(range(c, i, end(c))) >= n implies
     plus(c, i, n) = if n = 0 then i else plus(c, c++i, n - 1),
  i in range(c, begin(c), end(c)) and size(range(c, begin(c), i)) >= n implies
     minus(c, i, n) = if n = 0 then i else minus(c, c -- i, n - 1),
  i in range(c, begin(c), end(c)) and plus(c, i, n) < end(c) implies
     brackets(c, i, n) = c*plus(c, i, n),
  i in range(c, begin(c), end(c)) and j in range(c, begin(c), end(c)) implies
    (i < j) = ((for some n: naturals) plus(c, i, n) = j).
```

Used in part 1.

[[Having to have the container as an argument makes `+` and `-` ternary operations and thus infix notation is ruled out, which is an inconvenience.]]

10 Reversible Container Concept

⟨Reversible container concept 13b⟩ ≡

```
Definition: Reversible-container
refines Forward-container [with Bidirectional-iterator as Forward-iterator];
uses Bidirectional-iterator [with reverse-iterators as iterators];
introduces
  reverse-iterator(iterators) -> reverse-iterators,
  rbegin(containers) -> reverse-iterators,
  rend(containers) -> reverse-iterators;
requires (for c: containers; i: iterators; r: reverse-iterators)
  rbegin(c) = reverse-iterator(end(c)),
  rend(c) = reverse-iterator(begin(c)),
  r = reverse-iterator(i) implies c++r = reverse-iterator(c -- i)
       and c -- r = reverse-iterator(c++i).
```

Used in part 1.

Note that `c++r` and `c--r` are legal because of the `uses` clause.

⟨Some invariants of reversible containers 13c⟩ ≡

```
Lemma: Reversible-container
obeys (for c: containers)
  valid(range(c, rbegin(c), rend(c))),
  size(range(c, rbegin(c), rend(c))) = size(c).
```

Used in part 1.

11 Random Access Container Concept

⟨Random access container concept 14a⟩ ≡

```
Definition: Random-access-container
refines Reversible-container [with Random-access-iterator as
                                      Bidirectional-iterator];
introduces brackets(containers, naturals) -> objects;
requires (for d: nonempty-containers; n: naturals)
  n < size(d) implies brackets(d, n) = d*(plus(d, begin(d), n)).
```

Used in part 1.

12 Vector Concept

⟨Vector concept 14b⟩ ≡

```
Definition: Vector
refines Random-access-container [with vectors as containers,
                                    nonempty-vectors as nonempty-containers],
        Back-insertion-sequence [with vectors as sequences,
                                    nonempty-vectors as nonempty-sequences];
introduces capacity(vectors) -> naturals,
  reserve(vectors, naturals) -> vectors,
  usable(vectors, iterators) -> bool;
generates vectors freely using construct, push-back;
requires (for v, v1: vectors; w: nonempty-vectors; n: naturals;
              i, i1, j, k: iterators; e: elements)
  capacity(v) >= size(v),
  capacity(reserve(v, n)) >= n,
  v1 = reserve(v, n) implies
     size(v1) = size(v)
     and v1 access-equivalent-to v,
  n <= capacity(v) implies reserve(v, n) = v,
  usable(v, begin(v)),
  usable(v, end(v)),
  valid(range(v, i, j)) and
    usable(v, i) and usable(v, j) and k in range(v, i, j) implies usable(v, k),
  i in range(w, begin(w), end(w)) implies not(usable(erase(w, i), end(w))),
  i in range(w, begin(w), end(w)) and j in range(w, begin(w), end(w))
    and position(w, i) < position(w, j) and n = size(range(w, i, j))
    and k in range(w, minus(w, end(w), n), end(w)) implies
      not(usable(erase(w, i, j), k)).
```

Used in part 1.

13 Deque Concept

The deque concept simply combines three of the previously given concepts.

⟨Deque concept 14c⟩ ≡

```
Definition: Deque
refines Random-access-container [with deques as containers,
                                    nonempty-deques as nonempty-containers],
        Front-insertion-sequence [with deques as sequences,
                                    nonempty-deques as nonempty-sequences],
        Back-insertion-sequence [with deques as sequences,
                                    nonempty-deques as nonempty-sequences];
generates deques freely using construct, push-back.
```

Used in part 1.

14 List Concept

We first introduce a splice operation on iterator ranges:

⟨Spliceable container concept 15a⟩ ≡

```
Definition: Spliceable-container
refines Reversible-container;
introduces
  +(ranges, ranges) -> ranges,
  splice(containers, iterators, containers) -> containers,
  splice(containers, iterators, containers, iterators) -> containers,
  splice(containers, iterators, containers, iterators, iterators) -> containers;
requires (for c, d, e, u, v, w: containers; i, j, k, l, p, q: iterators)
  range(e, p, q) = range(c, i, j) + range(d, k, l)
    implies if i = j then e = d and p = k and q = l
            else range(e, p, j) = range(c, i, j) and e++(c -- j) = k
                 and range(e, k, l) = range(d, k, l),
  w = splice(u, p, v) implies
    range(w, begin(w), end(w)) =
      range(u, begin(u), p) + range(v, begin(v), end(v)) + range(u, p, end(u)),
  w = splice(u, p, v, i) implies
    range(w, begin(w), end(w)) =
      range(u, begin(u), p) + range(v, i, v++i) + range(u, p, end(u)),
  w = splice(u, p, v, i, j) implies
      range(w, begin(w), end(w)) =
        range(u, begin(u), p) + range(v, i, j) + range(u, p, end(u)).
```

Used in part 1.

[[The following definition should be broken down into smaller parts for easier understanding.]]

⟨List concept 15b⟩ ≡

```
Remark: Pragma: fundes, path=Predicate-set.
Definition: List
refines Front-insertion-sequence [with lists as sequences,
                                  nonempty-lists as nonempty-sequences],
  Back-insertion-sequence [with lists as sequences,
                                nonempty-lists as nonempty-sequences],
  Spliceable-container [with lists as containers,
                             nonempty-lists as nonempty-containers];
  uses Predicate-set [with elements as domain],
       Binary-predicate-set [with elements as domain1, elements as domain2],
       Total-order [with elements as domain];
introduces
  remove(lists, elements) -> lists,
  remove(lists, iterators, iterators, elements) -> lists (private),
  remove-if(lists, predicates) -> lists,
  remove-if(lists, iterators, iterators, predicates) -> lists (private),
  unique(lists) -> lists,
  unique(lists, iterators, iterators) -> lists,
  unique-if(lists, binary-predicates) -> lists,
  unique-if(lists, iterators, iterators, binary-predicates) -> lists,
  interleaving(lists, lists, lists) -> bool (private),
  interleaving(lists, iterators, iterators,
               lists, iterators, iterators,
               lists, iterators, iterators) -> bool (private),
  ordered(lists, binary-predicates) -> bool (private),
  ordered(lists, iterators, iterators, binary-predicates)
               -> bool (private),
```

```
  default-comparison :  -> binary-predicates,
  ordered(c: lists) = ordered(c, default-comparison),
  ordered(c: lists, i: iterators, j: iterators) =
    ordered(c, i, j, default-comparison),
  merge(lists, lists) -> lists,
  merge(lists, lists, binary-predicates) -> lists,
  count(lists, elements) -> naturals (private),
  count(lists, iterators, iterators, elements) -> naturals (private),
  sort(lists) -> lists,
  permutation(lists, lists) -> bool,
  sort(lists, binary-predicates) -> lists,
  reverse(lists) -> lists,
  reverse(lists, iterators, iterators) -> lists (private);
generates lists freely using construct, push-back;
requires (for u, v, w: nonempty-lists;
          e: elements; i, j, p, q, r, s: iterators;
              b: predicates; c: binary-predicates)
  i in range(u, begin(u), end(u)) and w = erase(u, i) implies
    range(w, begin(w), end(w)) =
       range(u, begin(u), i) + range(u, u++i, end(u)),
  remove(u, e) = remove(u, begin(u), end(u), e),
  valid(range(u, i, j)) implies
    remove(u, i, j, e) = if i = j then u
                         else if access(u*i) = e then
                            remove(erase(u, i), u++i, j, e)
                         else
                            remove(u, u++i, j, e),
  remove-if(u, b) = remove-if(u, begin(u), end(u), b),
  valid(range(u, i, j)) implies
    remove-if(u, i, j, b) = if i = j then u
                         else if value(b, access(u*i)) then
                            remove-if(erase(u, i), u++i, j, b)
                         else
                            remove-if(u, u++i, j, b),
  unique(u) = unique(u, begin(u), end(u)),
  valid(range(u, i, j)) implies
    unique(u, i, j) = if i = j then u else
                      if u++i = j then u else
                      if access(u*i) = access(u*(u++i))
                        then unique(erase(u, i), u++i, j)
                        else unique(u, u++i, j),
  unique-if(u, c) = unique-if(u, begin(u), end(u), c),
  valid(range(u, i, j)) implies
    unique-if(u, i, j, c) = if i = j then u else
                      if u++i = j then u else
                      if value(c, access(u*i), access(u*(u++i)))
                        then unique-if(erase(u, i), u++i, j, c)
                        else unique-if(u, u++i, j, c),
  reverse(u) = reverse(u, begin(u), end(u)),
  valid(range(u, i, j)) implies
    reverse(u, i, j) = if i = j then u
                       else splice(reverse(u, u++i, j), j, u, i),
  interleaving(u, v, w) = interleaving(u, begin(u), end(u),
                                       v, begin(v), end(v),
                                       w, begin(w), end(w)),
  valid(range(u, i, j)) and valid(range(v, p, q))
    and valid(range(w, r, s)) implies
    interleaving(u, i, j, v, p, q, w, r, s) =
      if i = j then
```

```
              if p = q then r = s else r != s and v*p = w*r and
                                         interleaving(u, i, j, v, v++p, q, w, w++r, s)
           else if p = q then r != s and u*i = w*r and
                                        interleaving(u, u++i, j, v, p, q, w, w++r, s)
           else r != s and (u*i = w*r and
                                interleaving(u, u++i, j, v, p, q, w, w++r, s)
                              or v*p = w*r and
                                   interleaving(u, i, j, v, v++p, q, w, w++r, s)),
       ordered(u) = ordered(u, begin(u), end(u)),
       valid(range(u, i, j)) implies
         ordered(u, i, j) =
            (i = j or u++i = j or not(access(u*(u++i)) < access(u*i))
                                  and ordered(u, u++i, j)),
       ordered(u, c) = ordered(u, begin(u), end(u), c),
       valid(range(u, i, j)) implies
         ordered(u, i, j, c) =
            (i = j or u++i = j or not(value(c, access(u*(u++i)), access(u*i)))
                                  and ordered(u, u++i, j, c)),
       w = merge(u, v) implies interleaving(u, v, w) and
             (ordered(u) and ordered(v) implies ordered(w)),
       w = merge(u, v, c) implies interleaving(u, v, w) and
             (ordered(u, c) and ordered(v, c) implies ordered(w, c)),
       count(u, e) = count(u, begin(u), end(u), e),
       valid(range(u, i, j)) implies
         count(u, i, j, e) = if i = j then 0
                             else if access(u*i) = e then 1 + count(u, u++i, j, e)
                                  else count(u, u++i, j, e),
       permutation(u, v) = ((for e: elements) count(u, e) = count(v, e)),
       w = sort(u) implies
          permutation(w, u) and ordered(w) and
          ((for i: iterators) i in range(w, begin(w), end(w)) implies w*i = u*i),
       w = sort(u, c) implies
          permutation(w, u) and ordered(w, c) and
          ((for i: iterators) i in range(w, begin(w), end(w)) implies w*i = u*i).
```

Used in part 1.

15 Associative Container Concepts

⟨Associative container concept 18a⟩ ≡

```
Precedence: nonassociative{=, has-key} < {*}.

Definition: Associative-container
refines Basic-sequence [with Bidirectional-iterator as Forward-iterator];
introduces count(sequences, elements) -> naturals,
  count(sequences, iterators, iterators, elements) -> naturals (private),
  find(sequences, elements) -> iterators,
  erase(sequences, elements) -> sequences,
  equal-range(sequences, elements) -> (iterators, iterators),
  has-key(objects, elements) -> bool,
  equiv(elements, elements) -> bool;
requires (for i, j: iterators; c: sequences; k: elements)
  count(c, k) = count(c, begin(c), end(c), k),
  valid(range(c, i, j)) implies
    count(c, i, j, k) = if i = j then 0 else
                        if c*i has-key k then 1 + count(c, c++i, j, k) else
                        count(c, c++i, j, k),
  equal-range(c, k) = (i, j) implies valid(range(c, i, j)) and
     if count(c, k) = 0 then
        i = end(c) and j = end(c)
     else
       ((for q: iterators) q in range(c, begin(c), end(c)) implies
           (c*q has-key k) = (q in range(c, i, j))),
  equal-range(c, k) = (i, j) and constant(c) implies
     points-to-constant(i) and points-to-constant(j),
  i = find(c, k) implies
      if count(c, k) = 0 then i = end(c)
      else i in range(c, begin(c), end(c)) and c*i has-key k
           and count(c, begin(c), i, k) = 0,
  constant(c) implies points-to-constant(find(c, k)),
  erase(c, k) = erase(c, i, j) where equal-range(c, k) = (i, j).
```

Used in part 1.

15.1 Sorted Versus Hashed Associative Containers

⟨Sorted associative container concept 18b⟩ ≡

```
Precedence: nonassociative{value-compare}.

Definition: Sorted-associative-container
refines Associative-container;
uses Strict-weak-order [with key-comp as <, elements as domain];
introduces
  value-compare(objects, objects) -> bool,
  lower-bound(sequences, elements) -> iterators,
  upper-bound(sequences, elements) -> iterators;
requires (for c: sequences; x, x1: objects; k, k1: elements; i, j: iterators)
// what's wrong with the following line?
// (x value-compare x1) = (k key-comp k1) where x has-key k and x1 has-key k1,
  (x value-compare x1) = key-comp(k, k1) where x has-key k and x1 has-key k1,
  ((for p, q: indices) p in range(least, least + size(c))
     and q in range(least, least + size(c)) and p < q
     implies not(key-comp(access(c, q), access(c, p)))),
  (k equiv k1) = (not(key-comp(k, k1)) and not(key-comp(k1, k))),
  lower-bound(c, k) = i where equal-range(c, i, j) = (i, j),
```

```
  upper-bound(c, k) = j where equal-range(c, i, j) = (i, j).
```

Used in part 1.

⟨Hashed associative container concept 19a⟩ ≡

```
Definition: Hashed-associative-container
refines Associative-container;
introduces hash(elements) -> naturals;
requires (for c: sequences; k, k1: elements)
  (k equiv k1) = (k = k1).
```

Used in part 1.

15.2 Multiple Versus Unique Associative Containers

⟨Multiple associative container concept 19b⟩ ≡

```
Definition: Multiple-associative-container
refines Associative-container;
introduces
  insert(sequences, elements) -> (sequences, iterators);
requires (for c: sequences; c1: nonempty-sequences;
          e: elements; i1: iterators)
  insert(c, e) = (c1, i1) implies
     size(c1) = size(c) + 1 and access(c1*i1) = e
     and ((for p: indices)
           (p in range(least, position(c1, i1))
               implies access(c1, p) = access(c, p))
           and (p in range(position(c1, i1), least + size(c))
               implies access(c1, next(p)) = access(c, p))).
```

Used in part 1.

⟨Unique associative container concept 19c⟩ ≡

```
Definition: Unique-associative-container
refines Associative-container;
introduces
  insert(sequences, elements) -> (sequences, iterators, bool);
requires (for c: sequences; c1: nonempty-sequences; b: bool;
         e: elements; i, i1: iterators)
  count(c, e) = 0 or count(c, e) = 1,
  insert(c, e) = (c1, i1, b) implies
   b = (find(c, e) = end(c)) and
   if b then
     size(c1) = size(c) + 1 and access(c1*i1) = e
     and ((for p, q: indices)
           (p in range(least, position(c1, i1))
               implies access(c1, p) = access(c, p))
           and (q in range(position(c1, i1), least + size(c))
               implies access(c1, next(q)) = access(c, q)))
   else
     c1 = c.
```

Used in part 1.

15.3 Simple Versus Pair Associative Containers

⟨Simple associative container concept 19d⟩ ≡

```
Definition: Simple-associative-container
refines Associative-container;
requires (for i: iterators; x: objects; k: elements)
  points-to-constant(i),
  (x has-key k) = (access(x) equiv k).
```

Used in part 1.

⟨Pair and associative-pair concepts 20a⟩ ≡

```
Abbreviation: First is Assignable
  [with first-objects as objects, first-elements as elements].

Abbreviation: Second is Assignable
  [with second-objects as objects, second-elements as elements].

Definition: Pair
refines Assignable [with pairs as objects];
uses First, Second;
introduces construct(first-elements, second-elements) -> pairs,
  first(pairs) -> first-objects,
  second(pairs) -> second-objects;
requires (for f: first-elements; s: second-elements)
  access(first(construct(f, s))) = f,
  access(second(construct(f, s))) = s.

Abbreviation: Key is Assignable
  [with first-objects as objects, keys as elements].

Remark: Pragma: path=Assignable.
Definition: Associative-pair
refines Pair [with Key as First, keys as first-elements];
requires (for p: pairs)
  constant(first(p)).
Pragma: path=().
```

Used in part 1.

⟨Pair associative container concept 20b⟩ ≡

```
Definition: Pair-associative-container
refines Associative-container [with Associative-pair as Accessible,
                                    pairs as objects, keys as elements];
requires (for x: pairs; k: keys)
 (x has-key k) = (access(first(x)) equiv k).
```

Used in part 1.

15.4 Combinations of Associative Container Concepts

⟨Combinations of associative container concepts 20c⟩ ≡

```
Definition: Set-associative-container
refines Simple-associative-container, Unique-associative-container,
        Sorted-associative-container.

Definition: Multiset-associative-container
refines Simple-associative-container, Multiple-associative-container,
        Sorted-associative-container.

Definition: Map-associative-container
refines Pair-associative-container, Unique-associative-container,
        Sorted-associative-container;
introduces brackets(sequences, keys) -> second-objects;
requires (for m, m1: sequences; i: iterators; k: keys; b: bool)
  brackets(m, k) = second(m1*i)
       where insert(m, construct(k, construct-element)) = (m1, i, b).

Definition: Multimap-associative-container
```

```
refines Pair-associative-container, Multiple-associative-container,
        Sorted-associative-container.

Definition: Hashed-set-associative-container
refines Simple-associative-container, Unique-associative-container,
        Hashed-associative-container.

Definition: Hashed-multiset-associative-container
refines Simple-associative-container, Multiple-associative-container,
        Hashed-associative-container.

Definition: Hashed-map-associative-container
refines Pair-associative-container, Unique-associative-container,
        Hashed-associative-container.

Definition: Hashed-multimap-associative-container
refines Pair-associative-container, Multiple-associative-container,
        Hashed-associative-container.
```
Used in part 1.

[[Need to add to the above concepts the definition of iterator validity, and to expand the discussion.]]

References

[1] Preston Briggs. Nuweb, a simple literate programming tool. 1989. 1

[2] International Organization for Standardization (ISO). *ISO/IEC Final Draft International Standard 14882: Programming Language C++*, 1998. 1

[3] Deepak Kapur and David Musser. Tecton: A framework for specifying and verifying generic system components. Technical report, RPI Computer Science Department Technical Report 92-20, July 1992. 1

[4] Deepak Kapur, David Musser, and Alexander Stepanov. Operators and algebraic structures. In *Proc. of Conference on Functional Programming Languages and Computer Architecture, Portsmuth, New Hampshire*, October 1981. 1

[5] Donald E. Knuth. Literate programming. *Computer Journal*, 27:97–111, 1984. 1

[6] Rüdiger Loos, David R. Musser, Sibylle Schupp, and Christoph Schwarzweller. Tecton Concept Library. Technical Report WSI-99-2, Fakultät für Informatik, Universität Tübingen, January, updated July 1999. 3, A

[7] David Musser. The Tecton Concept Description Language, July 1998. 1

[8] David Musser and Atul Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison-Wesley, 1996. 1

[9] SGI. SGI standard template library programming guide. `http://www.sgi.com/tech/stl/`, 1998. 1, 4

[10] Alexander Stepanov and M. Lee. The Standard Template Library. Technical report, HPL-94-34, April 1994. revised July 7, 1995. 1

A Basic concepts from algebra

The following is a listing of the basic concepts from algebra that are used in the present specification. A documented version of these and other algebra concepts is available in [6].

⟨Through Natural 22⟩ ≡

```
Precedence: confix{_,}.

Definition: Boolean
  introduces bool,
    true  -> bool,
    false -> bool;
  generates bool freely using true, false.

Precedence: nonassociative{=, !=}.
Precedence: {implies} < {or, xor} < {and}
                < prefix{not} < nonassociative{=} < {:}.
Precedence: confix{(, ,, )}.

Extension: Boolean
  introduces
    not     : bool -> bool,
    and     : bool x bool -> bool,
    or      : bool x bool -> bool,
    xor     : bool x bool -> bool,
    implies : bool x bool -> bool;
  requires (for x, y: bool)
    (not true)    = false,
    (not false)   = true,
    (true and x)  = x,
    (false and x) = false,
    (x or y)      = (not (not x and not y)),
    (x xor y)     = (not x = y),
    (x implies y) = (not x or y).

Definition: Domain
  uses Boolean;
  introduces domain.

Precedence: nonassociative{=, R} < prefix{P}.

Definition: Unary-relation
  refines Domain;
  introduces P : domain -> bool.

Precedence: nonassociative{=} < nonassociative{in}.

Definition: Set
  uses Domain;
  introduces sets,
    empty-set: -> sets,
    in: domain x sets -> bool;
  requires
    (for a: domain) a in empty-set = false.

Precedence: nonassociative{in, into}.
Precedence: nonassociative{=} = {union} < {intersection} < {subset}.

Extension: Set
```

```
introduces
  nonempty-sets < sets,
  subset       : sets x sets -> bool,
  is-empty     : sets -> bool,
  complement   : sets -> sets,
  singleton    : domain -> sets,
  into         : domain x sets -> sets,
  union        : sets x sets -> sets,
  intersection : sets x sets -> sets;
requires (for d, e: domain; s, s1, s2: sets)
  (s1 subset s2) = (d in s1 implies d in s2),
  is-empty(s) = (s = empty-set),
  (d in (e into s1)) = ((d = e) or d in s1),
  (d in complement(s)) = (not d in s),
  singleton(d) = (d into empty-set),
  (s1 union empty-set) = s1,
  (s1 union (d into s2)) =
     if d in s1 then s1 union s2
                else d into (s1 union s2),
  (s1 intersection empty-set) = empty-set,
  (s1 intersection (d into s2)) =
     if d in s1 then d into (s1 intersection s2)
                else s1 intersection s2,
  s : nonempty-sets = (s != empty-set).

Remark: Lemma: Set
obeys (for d, e: domain; s, s1, s2: sets)
  is-empty(empty-set),
  (s subset empty-set) implies (s = empty-set),
  (d in singleton(e)) = (d = e),
  (d in (s1 union s2)) = ((d in s1) or (d in s2)),
  (d in (s1 intersection s2)) = ((d in s1) and (d in s2)).

Definition: Range
  uses Domain [with range as domain].



Precedence: nonassociative{=} < {*}.

Definition: Binary-op
  uses Domain;
  introduces * : domain x domain -> domain.

Precedence: nonassociative{R, <=}.

Definition: General-binary-relation
  uses Domain, Range;
  introduces R : domain x range -> bool.

Precedence: nonassociative{=} < {|}.

Definition: Right-regular
  refines Binary-op;
  introduces | : domain x domain -> bool;
  requires (for x, y: domain)
    x | y = (for some d: domain) x * d = y.

Definition: Right-identity
```

```
  refines Binary-op;
  introduces 1 -> domain;
  requires (for x: domain)
    x * 1 = x.

Precedence: nonassociative{=} < {+, -} < prefix{-, +} < {*}.

Definition: Right-distributive
  refines Binary-op, Binary-op [with + as *];
  requires (for x, y, z: domain)
    (x + y) * z = x * z + y * z.

Precedence: nonassociative{=} < {|}.

Definition: Left-regular
  refines Binary-op;
  introduces | : domain x domain -> bool;
  requires (for x, y: domain)
    x | y = (for some d: domain) d * x = y.

Definition: Left-identity
  refines Binary-op;
  introduces 1 -> domain;
  requires (for x: domain)
    1 * x = x.

Definition: Left-distributive
  refines Binary-op, Binary-op [with + as *];
  requires (for x, y, z: domain)
    x * (y + z) = x * y + x * z.

Definition: Commutative
  refines Binary-op;
  requires (for x, y: domain)
    x * y = y * x.

Definition: Associative
  refines Binary-op;
  requires (for x, y, z: domain)
    x * (y * z) = (x * y) * z.

Definition: Function
  refines General-binary-relation;
  introduces f: domain -> range;
  requires (for x: domain; y, y': range)
    (f(x) = y) = (x R y),
    f(x) = y and f(x) = y' implies y = y'.

Definition: Binary-relation
  refines Domain;
  introduces R : domain x domain -> bool.

Remark: Lemma: Binary-relation is General-binary-relation.

Precedence: prefix{-} < {*} < postfix{^(-1)}.

Definition: Right-inverses
  refines Right-identity, Right-regular;
  introduces ^(-1) : domain -> domain;
```

```
  requires (for x: domain)
    x * x^(-1) = 1.

Remark: Lemma: Right-inverses implies Right-regular.

Definition: Regular
  refines Left-regular, Right-regular.

Precedence: prefix{-} < {*} < postfix{^(-1)}.

Definition: Left-inverses
  refines Left-identity, Left-regular;
  introduces ^(-1) : domain -> domain;
  requires (for x: domain)
    x^(-1) * x = 1.

Remark: Lemma: Left-inverses implies Left-regular.

Definition: Identity
  refines Left-identity, Right-identity.

Definition: Distributive
  refines Left-distributive, Right-distributive.

Abbreviation: Semigroup is Associative.

Definition: Surjection
  refines Function;
  requires (for y: range) (for at least 1 x: domain)
    f( x) = y.

Definition: Injection
  refines Function;
  requires (for y: range) (for at most 1 x: domain)
    f(x) = y.

Definition: Transitive
  refines Binary-relation;
  requires
    (for x, y, z: domain) x R y and y R z implies x R z.

Definition: Symmetric
  refines Binary-relation;
  requires
    (for x, y: domain) x R y implies y R x.

Definition: Reflexive
  refines Binary-relation;
  requires
    (for x: domain) x R x.

Definition: Irreflexive
  refines Binary-relation;
  requires
    (for x: domain) not x R x.

Definition: Antisymmetric
  refines Binary-relation;
  requires
```

```
    (for x, y: domain) x R y and y R x implies x = y.

Definition: Inverses
  refines Left-inverses, Right-inverses.

Remark: Lemma: Inverses implies Regular.

Precedence: {/, *}.

Extension: Inverses
  introduces  / : domain x domain -> domain;
  requires (for x, y:domain)
    x/y = x * y^(-1).

Definition: Semigroup-homomorphism
  refines Semigroup, Semigroup [with image as domain];
  introduces
    h : domain -> image;
  requires (for x, y: domain)
    h(x*y) = h(x)*h(y).

Definition: Regular-semigroup
  refines Regular, Semigroup.

Definition: Monoid
  refines Semigroup, Identity.

Definition: Bijection
  refines Surjection, Injection.

Definition: Equivalence-relation
  refines Reflexive, Symmetric, Transitive.

Precedence: nonassociative{R, <}.

Definition: Strict-partial-order
  refines Irreflexive [with < as R],
          Transitive [with < as R].

Definition: Semigroup-monomorphism
  refines
    Semigroup-homomorphism,
    Injection [with h as f, image as range].

Definition: Semigroup-epimorphism
  refines
    Semigroup-homomorphism,
    Surjection [with h as f, image as range].

Precedence: nonassociative{=} < {+, -}.

Definition: Commutative-semigroup
  refines Regular-semigroup [with + as *],
          Commutative [with + as *].

Definition: Group
  refines Monoid, Inverses.

Definition: Abelian-monoid
```

```
  refines Monoid, Commutative.

Precedence: nonassociative{in, into}.
Precedence: nonassociative{=, equiv}.

Definition: Equivalence-class
  uses Set, Equivalence-relation [with equiv as R];
  introduces equivalence-classes,
    in : domain x equivalence-classes -> bool,
    equivalence-class : domain -> equivalence-classes;
  requires (for x, y: domain; e: equivalence-classes)
    (equivalence-class(x) = equivalence-class(y)) = (x equiv y),
    x in e = (equivalence-class(x) = e).

Precedence: nonassociative{R, <=, =}.

Definition: Partial-order
  refines Reflexive [with <= as R],
          Antisymmetric [with <= as R],
          Transitive [with <= as R].

Precedence: nonassociative{<, <=, >=, >, =}.
Extension: Partial-order
  introduces
    <  : domain x domain -> bool,
    >  : domain x domain -> bool,
    >= : domain x domain -> bool;
  requires (for x, y: domain)
    (x < y)  = (x <= y and x != y),
    (x > y)  = (not x <= y),
    (x >= y) = (x > y or x = y).

Remark: Lemma: Partial-order implies Strict-partial-order.

Definition: Semiring
  refines Commutative-semigroup, Semigroup, Distributive.

Definition: Trivial-group
  refines Group;
  requires (for x: domain) x = 1.

Definition: Group-of-order-2
  refines Group;
  requires (for x: domain) x * x = 1.

Remark: Lemma: Group-of-order-2 is Commutative.

Definition: Commutative-group
  refines Commutative, Group.

Definition: Set-of-representatives
  uses Equivalence-class;
  introduces
    set-of-representatives < domain,
    representative : equivalence-classes  -> domain,
    representative : domain -> domain;
  requires (for x: domain; e: equivalence-classes)
    x: set-of-representatives = (representative(x) = x),
    equivalence-class(representative(e)) = e,
```

```
    representative(x) = representative(equivalence-class(x)).

Definition: Total-order
  refines Partial-order;
  requires (for x, y: domain)
    x <= y or y <= x.

Definition: Trichotomy
  refines Strict-partial-order;
  requires (for x, y : domain)
    x < y or x = y or y < x.

Remark: Lemma: Trichotomy is Total-order .

Definition: Nondense-order
  refines Total-order;
  requires
    not ((for x, y: domain)
           x < y implies (for some z: domain) x < z and z < y).

Definition: Natural
  refines
    Semiring [with naturals as domain],
    Identity [with naturals as domain],
    Identity [with naturals as domain, + as *, 0 as 1],
    Commutative [with naturals as domain],
    Nondense-order [with naturals as domain];
  introduces
    next : naturals -> naturals,
       - : naturals x naturals -> naturals;
  generates naturals freely using 0, next;
  requires (for n, m: naturals)
     n + next(m) = next(n + m),
     n - 0 = n,                    0 - n = 0,
     next(n) - next(m) = n - m,    1 = next(0),
     n * 0 = 0,                    m * next(n) = m * n + 1,
     0 <= n,                       next(n) > 0,
     (next(m) <= next(n)) = (m <= n).
```

Used in part 3a.