The Tecton Concept Description Language

David R. Musser*
Rensselaer Polytechnic Institute
Troy, New York 12180
musser@cs.rpi.edu

*This work was partially supported by the Wilhelm Schickard Institut für Informatik, Universität Tübingen.

Tecton concept description language

- A small language for describing and using abstract concepts in software development—particularly generic programming
- Simple syntax
- Relatively simple semantics—set theoretic
- Supports inheritance and instantiation in a fundamental way
- Includes a *realization* construct—for linking to efficient implementations in real languages
- No explicit parameter lists: all constituents can be replaced—provided sort correctness and semantic consistency are preserved

A first example: STL container concept expressed as a Tecton definition

⟨Container concept 3⟩ ≡

```
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 25.

The five kinds of Tecton sentences

- Definitions, extensions, abbreviations, lemmas, realizations.
- Each of the sentence forms introduces or refers to a concept name, called the *subject* of the sentence.
- A definition, abbreviation, or realization defines its subject: it associates the concept name with a meaning, which is a concept—i.e., a set of algebras.
- An extension adds information about new identifiers to a concept while preserving the meaning it gives to other identifiers.
- Lemmas assert that certain properties follow from the definition of and extensions to the concept.

⟨Basic concepts 5⟩ ≡

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

Definition: Set uses Boolean; introduces domain.

Definition: Natural-number refines Set [with naturals as domain];
introduces 0 -> naturals,           1 -> naturals,
       next(naturals) -> naturals,  +(naturals, naturals) -> naturals,
       -(naturals, naturals) -> naturals,
       *(naturals, naturals) -> naturals;
generates naturals freely using 0, next;
requires (for n, m: naturals)
       n + 0 = n,                       n + next(m) = next(n + m),
       n - 0 = n,                       0 - n = 0,
       next(n) - next(m) = n - m,       1 = next(0),
       i * 0 = 0,                       i * next(j) = i * j + i.
```

Used in part 25.

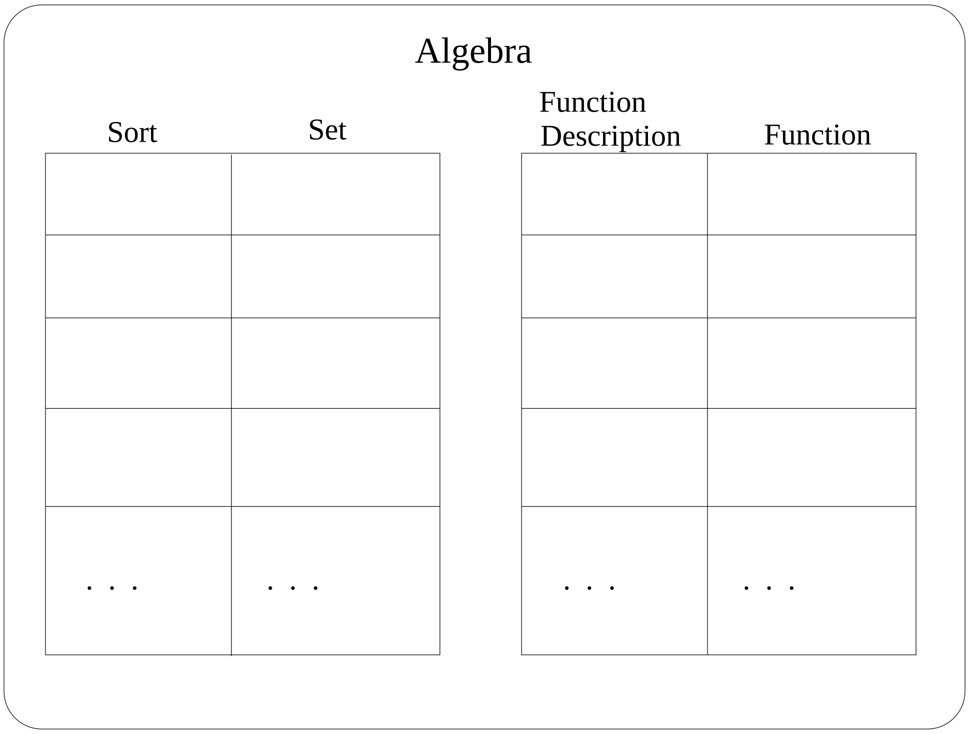
Semantics of a Tecton concept description

- A *concept* is a set of (many-sorted) algebras.
- An *algebra* a pair of mappings:
  - A mapping from *sorts* to sets.
  - A mapping from *function descriptions* to (mathematical) functions.
- A *sort* is an identifier.
- A *function description* has the form

  function-identifier `(` list of *sort*s `)` `->` list of *sort*s

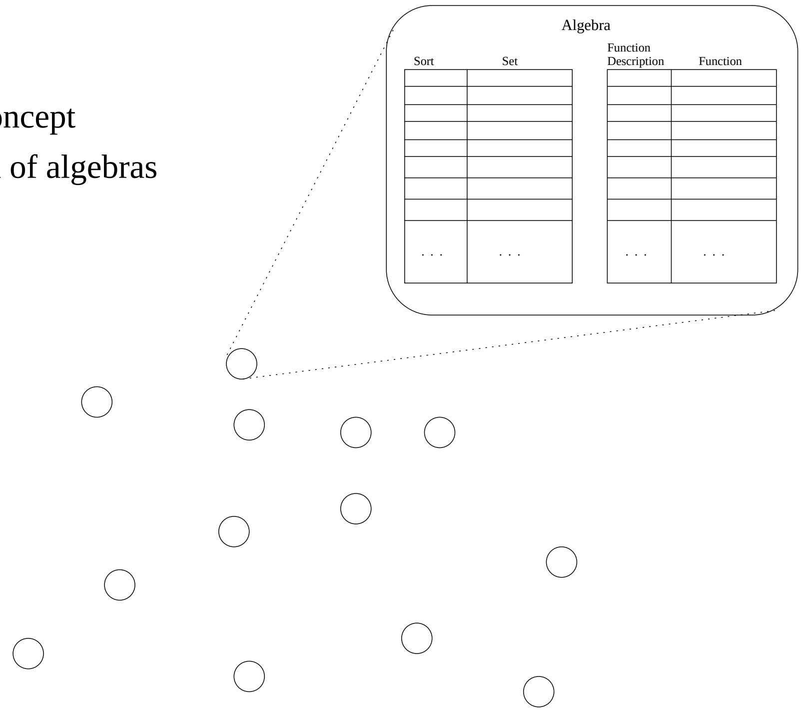
A Tecton concept description specifies a concept in terms of requirements that an algebra must satisfy in in order to belong to the concept.

Algebra — a pair of mappings

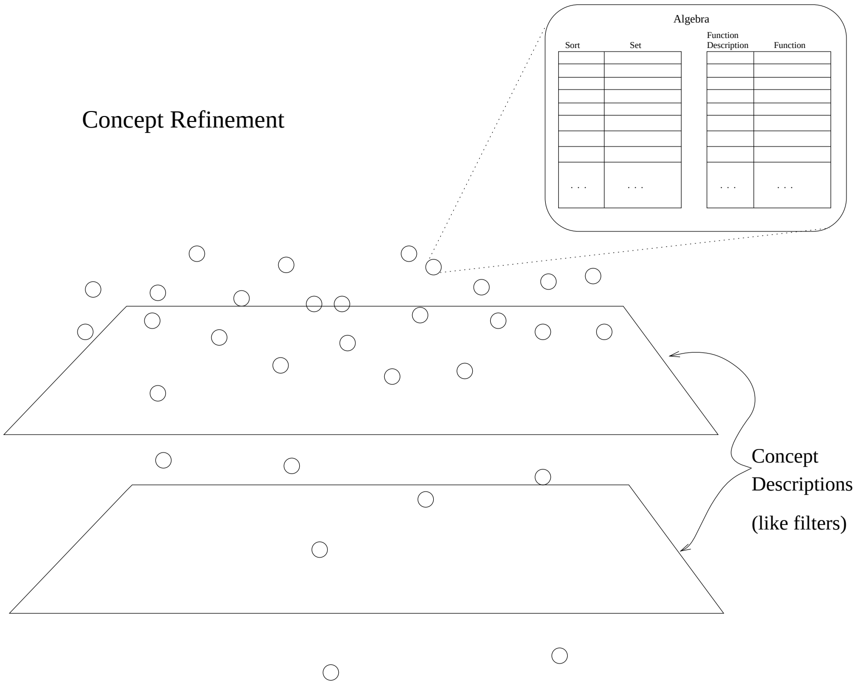


Concept — a collection of algebras

A (Tecton) concept
is a collection of algebras



Concept description — a *filter*
(admits some algebras, rejects others)



Defining relation concepts in Tecton

⟨Relation concepts 10⟩ ≡

```
Definition: Binary-relation refines Set;
  introduces R(domain, domain) -> bool.

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: Symmetric refines Binary-relation;
  requires (for x, y: domain) x R y implies y R x.
Definition: Anti-symmetric refines Binary-relation;
  requires (for x, y: domain) (x R y and y R x) implies 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: Equivalence-relation
  refines Reflexive, Symmetric, Transitive.
```

Used in part 27.

Extending existing concept descriptions (and stating Lemmas)

⟨Examples of Tecton extension sentences 11⟩ ≡

```
Extension: Natural-number introduces <=(naturals, naturals) -> bool;
requires (for n, m: naturals)
       0 <= n,   not(next(n) <= 0),   (next(m) <= next(n)) = (m <= n).

Extension: Natural-number
introduces <(x: naturals, y: naturals) = (x <= y and not(x = y)),
           >=(x: naturals, y: naturals) = not(x < y),
           >(x: naturals, y: naturals) =  not(x <= y).

Lemma: Natural-number implies Transitive [with < as R].
```

Used in part 27.

Defining order concepts abstractly in Tecton

⟨Order concepts 12⟩ ≡

```
Definition: Strict-partial-order
refines Irreflexive, Transitive.

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

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

Used in part 27.

More order concepts

⟨More order concepts 13⟩ ≡

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

Definition: Strict-total-order refines Strict-weak-order
   requires (for x, y: domain)
     equivalent(x, y) = (x = y).

Lemma: Natural-number implies Strict-total-order.
```

Used in part 27.

The Standard Template Library

- STL is a general-purpose library of generic algorithms and data structures communicating through iterators
- Contains many small components that can be plugged together and used in an application
- Provides a framework into which different programming problems can be decomposed
- STL pillars: efficiency, orthogonality, and a solid theoretical foundation

How solid is the theoretical foundation of STL?

- Semantics is given only informally as a set of requirements stated in English.
- Incomplete, depends heavily on reader's (and library implementor's) prior knowledge and "good intentions"
- Using Tecton to express the semantics formally is one answer

Sequence concept

The sequence concept is an refinement of a basic sequence concept (which in turn is a refinement of the container concept) with functions for inserting elements and getting the front element.

⟨Sequence concept 16⟩ ≡

```
Definition: Sequence
refines Basic-sequence;
introduces
  insert(sequences, iterators, elements) -> (sequences, iterators),
  insert(sequences, sequences, iterators, iterators) -> sequences,
  front(nonempty-sequences) -> elements;
requires (for c, c1, c2: sequences; d, d1: nonempty-sequences;
          i, i1, j, k: iterators; e: elements; p, q: indices; n: naturals)
  ⟨Axiom for inserting one element 17⟩
  ⟨Axiom for inserting a range of elements ?⟩
  front(d) = d*begin(d).
```

Used in part 25.

Insertion properties

⟨Axiom for inserting one element 17⟩ ≡

```
(c1, i1) = insert(c, i, e) implies
   size(c1) = size(c) + 1 and access(c1*i1) = e
   and position(c1, i1) = position(c, i)
   and ((for p, q: indices)
         (p in range(least, position(c, i))
             implies access(c1, p) = access(c, p))
         and (q in range(position(c, i), least + size(c))
             implies access(c1, next(q)) = access(c, q))),
```

Used in part 16.

Front insertion sequence concept

⟨Front insertion sequence concept 18⟩ ≡

```
Definition: Front-insertion-sequence
refines Sequence;
introduces
  push-front(sequences, objects) -> 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 25.

Back insertion sequence concept

⟨Back insertion sequence concept 19⟩ ≡

```
Definition: Back-insertion-sequence refines Sequence;
introduces
  back(nonempty-sequences) -> objects,
  push-back(sequences, objects) -> 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 25.

Bidirectional iterator concept

⟨Bidirectional iterator concept 20⟩ ≡

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

Used in part 25.

Random access iterator concept

⟨Random-access iterator concept 21⟩ ≡

```
Definition: Random-access-iterator
refines Bidirectional-iterator, Less-than-comparable;
introduces +(containers, iterators, naturals) -> iterators,
  -(containers, iterators, naturals) -> iterators,
  brackets(containers, iterators, naturals) -> objects;
requires (for c: containers; d: nonempty-containers; i, j: iterators;
          n: Naturals)
  i in range(c, begin(c), end(c)) and size(range(c, i, end(c)) >= n implies
     +(c, i, n) = if n = 0 then i else +(c, c++i, n-1),
  i in range(c, begin(c), end(c)) and size(range(c, begin(c), i) >= n implies
     -(c, i, n) = if n = 0 then i else -(c, c--i, n-1),
  i in range(c, begin(c), end(c)) and i + n < end(d) implies
     brackets(d, i, n) = d*(+(d, 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) i + n = j).
```

Used in part 25.

Reversible container concept

⟨Reversible container concept 22⟩ ≡

```
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 = c--i and c--r = c++i.
```

Used in part 25.

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

Random access container concept

⟨Random access container concept 23⟩ ≡

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

Used in part 25.

Deque concept

⟨Deque concept 24⟩ ≡

```
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 25.

Outline of STL container and iterator formalization

"`container.tec`" 25 $\equiv$

$\langle$Basic concepts 5$\rangle$
$\langle$Basic-container concept ?$\rangle$
$\langle$Trivial iterator concept ?$\rangle$
$\langle$Input iterator concept ?$\rangle$
$\langle$Output iterator concept ?$\rangle$
$\langle$Container concept 3$\rangle$
$\langle$Forward iterator concept ?$\rangle$
$\langle$Extension of forward iterator concept ?$\rangle$
$\langle$Constness extensions of forward iterator concept ?$\rangle$
$\langle$Forward container concept ?$\rangle$
$\langle$Range equivalence concept ?$\rangle$
$\langle$Access equivalence concept ?$\rangle$
$\langle$Lexicographical order concept ?$\rangle$
$\langle$Indexing concepts ?$\rangle$
$\langle$Basic sequence concept ?$\rangle$
$\langle$Sequence concept 16$\rangle$
$\langle$Default construction lemma ?$\rangle$
$\langle$Front insertion sequence concept 18$\rangle$
$\langle$Back insertion sequence concept 19$\rangle$
$\langle$Bidirectional iterator concept 20$\rangle$
$\langle$Random-access iterator concept 21$\rangle$

Stuff

`"stuff.tec"` 27 $\equiv$

> ⟨Relation concepts 10⟩
> ⟨Examples of Tecton extension sentences 11⟩
> ⟨Order concepts 12⟩
> ⟨More order concepts 13⟩