Report on the Larch Shared Language
Version 2.3

J.V. Guttag, J.J. Horning, and A. Modet

April 14, 1990

SRC Research Report 58

Authors' Abstract

The Larch family of languages is used to specify program interfaces in a two-tiered definitional style. Each Larch specification has components written in two languages: one that is designed for a specific programming language and another that is independent of any programming language. The former are the *Larch interface languages*, and the latter is the *Larch Shared Language (LSL)*. Version 2.3 of LSL is similar to previous versions, but contains a number of refinements based on experience writing specifications and developing tools to support the specification process. This report contains an informal introduction and a self-contained language definition.

This report supersedes Pieces II and III of *Larch in Five Easy Pieces* [Guttag, Horning, and Wing 1985b] and "Report on the Larch Shared Language" [Guttag and Horning 1986].

Report on the Larch Shared Language, Version 2.3

Chapter 1

Overview

1.1. Introduction

The Larch family of specification languages supports a two-tiered definitional approach to specification [Guttag, Horning, and Wing 1985a]. Each specification has components written in two languages: one designed for a specific programming language and another independent of any programming language. The former are called *Larch interface languages*, and the latter the *Larch Shared Language* (*LSL*).

Larch interface languages are used to specify the interfaces between program components. Each specification provides the information needed to use the interface and to write programs that implement it. A critical part of each interface is how the component communicates with its environment. Communication mechanisms differ from programming language to programming language, sometimes in subtle ways. It is easier to be precise about communication when the interface specification language reflects the programming language. Specifications written in such interface languages are generally shorter than those written in a "universal" interface language. They are also clearer to programmers who implement components and to programmers who use them.

Each Larch interface language deals with what can be observed about the behavior of components written in a particular programming language. It incorporates programming-language-specific notations for features such as side effects, exception handling, iterators, and concurrency. Its simplicity or complexity depends largely upon the simplicity or complexity of the observable state and state transformations of its programming language. For example, an interface specification for a window system procedure to be implemented in CLU [Liskov and Guttag 1986] might be

> `addWindow` = **proc** (v: `View`, w: `Window`, c: `Coord`) **signals** (`duplicate`)
>   **modifies** v
>   **ensures** v_{post} = `addW`(v, w, c)
>   **except when** $w \in v$ **signals** `duplicate` **ensures** $v_{post} = v$

To understand such a specification, it is necessary to know both the meanings of the interface language constructs (e.g., **proc**, **signals**, **modifies**) and the meanings of operators appearing in expressions (e.g., `addW`, $\in$). Larch Shared Language specifications are used to define the latter. Specifiers are not limited to a fixed set of operators, but can use LSL to create specialized vocabularies suitable for particular interface specifications. An LSL specification that defined the meaning of `addW` and $\in$ could be used to give precise answers to questions such as what it means for a window to be in a view (visible or possibly obscured?), or what it means to add a window to a view that may contain other windows at the same location.

Larch encourages a separation of concerns, with mathematical abstractions in the LSL tier, and programming pragmatics in the interface tier. We encourage specifiers to keep the difficult parts in the LSL tier, for several reasons:

- LSL abstractions are more likely to be reusable than interface specifications.
- LSL has a simpler underlying semantics than most programming languages (and hence than most interface languages), so that specifiers are less likely to make mistakes.
- It is easier to make and check claims about semantic properties of LSL specifications than about semantic properties of interface specifications.

This chapter is an informal introduction to the Larch Shared Language, Version 2.3. It introduces all the features of the language, briefly discusses how they are intended to be used, and closes with a reference grammar. The following chapter is a rigorous definition of the language.

1.2. Simple Algebraic Specifications

LSL's basic unit of specification is a *trait*. A trait may describe an abstract data type or may encapsulate a property shared by several data types. Consider the following specification of tables that store values in indexed places:

```
Table: trait
  introduces
    new: → Tab
    add: Tab, Ind, Val → Tab
    __∈__: Ind, Tab → Bool
    lookup: Tab, Ind → Val
    isEmpty: Tab → Bool
    size: Tab → Card
  asserts ∀ i, i': Ind, val: Val, t: Tab
    lookup(add(t, i, val), i') == if i = i' then val else lookup(t, i')
    ¬(i ∈ new)
    i ∈ add(t, i', val) == i = i' ∨ i ∈ t
    size(new) == 0
    size(add(t, i, val)) == if i ∈ t then size(t) else size(t) + 1
    isEmpty(t) == size(t) = 0
```

This is similar to a conventional algebraic specification [Bidoit 1988; Dahl, Langmyhr, and Owe 1986; Gaudel 1985; Guttag and Horning 1978; Wirsing 1989]. The part of the specification following **introduces** declares a list of *operators* (function identifiers), each with its *signature* (the *sorts* of its domain and range). Every operator used in a trait must be declared; the signatures are used to sort-check *terms* (expressions) in much the same way as function calls are type-checked in programming languages. The remainder of this specification constrains the operators by means of equations.

An equation consists of two terms of the same sort, separated by ==. Equations of the form *term* == `true` can be abbreviated by simply writing the term; thus the second equation in the trait above is an abbreviation for $\neg(i \in \texttt{new})$ == `true`.

The characters "__" in an operator declaration indicate that the operator will be used in mixfix expressions. For example, $\in$ is declared as a binary infix operator. Infix, prefix, postfix, and distributed operators are integral parts of many familiar notations, and their use can contribute substantially to the readability of specifications. LSL's grammar for mixfix terms is intended to ensure that legal terms parse as readers expect—even without studying the grammar. Writers of specifications should study the grammar in Section 1.13—although fully parenthesized terms are always acceptable.[1]

The name of a trait is independent of the names that appear within it. In particular, we do not use sort identifiers to name units of specification. A trait need not correspond to an abstract data type, and often does not.

Each trait defines a *theory* (a set of formulas without free variables) in typed first-order logic with equality. Each theory contains the trait's assertions, the conventional axioms of first-order logic, everything that follows from them, and nothing else. This interpretation guarantees that the formulas in the theory follow only from the presence of assertions in the trait—never from their absence. This is in contrast to algebraic specification languages based on initial or final algebras [Ehrig and Mahr 1985; Goguen, Thatcher, and Wagner 1978; Sanella and Tarlecki 1987; Wand 1979]. Our interpretation is essential

[1] LSL has a very simple precedence scheme for operators: postfix operators consisting of a period followed by an identifier bind most tightly. Other user-defined operators and the built-in Boolean negation operator ($\neg$) bind more tightly than the built-in in equational operators ($=$ and $\neq$), which bind more tightly than the built-in Boolean connectives ($\wedge$, $\vee$, and $\Rightarrow$), which bind more tightly than ==. For example, the term $x + w\,.\texttt{a}.\texttt{b} = y \vee z$ is equivalent to $((x + ((w\,.\texttt{a}).\texttt{b})) = y) \vee z$. LSL allows unparenthesised infix terms with multiple operators at the same precedence level only if they are the same; it associates such terms from left to right. Thus $x \wedge y \wedge z$ is equivalent to $(x \wedge y) \wedge z$, but $x \vee y \wedge z$ isn't allowed.

to ensure that all theorems proved about an incomplete specification remain valid when it is completed.

LSL requires that each trait be *consistent:* it must not define a theory containing the equation `true` == `false`. Consistency is often difficult to prove, and is undecidable in general. But inconsistencies are often easy to detect [Garland, Guttag, and Horning 1990], and can be a useful indication that there is something wrong with a trait.

1.3. Getting Richer Theories

Equational theories are useful, but a stronger theory is often needed, for example, when specifying an abstract data type. The constructs **generated by** and **partitioned by** provide two ways of strengthening equational specifications.

A **generated by** clause asserts that all values of a sort can be generated by a given list of operators, thus providing a "generator induction" schema for the sort. For example, the natural numbers are generated by `0` and `successor`, and the integers are generated by `0`, `successor`, and `predecessor`.

The axiom "`Tab` **generated by** `new`, `add`", if added to `Table`, could be used to prove theorems by induction over `new` and `add`, such as

$$\forall\ t\text{: } \mathtt{Tab}\ \Big(\ \mathtt{isEmpty}(t) \vee \exists\ i\text{: } \mathtt{Ind}\ (\ i \in t\ )\Big)$$

A **partitioned by** clause asserts that all distinct values of a sort can be distinguished by a given list of operators. Terms that are not distinguishable using any of the partitioning operators of their sort are equal. For example, sets are partitioned by $\in$, because sets that contain the same elements are equal.

The axiom "`Tab` **partitioned by** $\in$, `lookup`", if added to `Table`, could be used to derive theorems that do not follow from the equations alone, such as

$$\forall\ t\text{: } \mathtt{Tab},\ i,\ i'\text{: } \mathtt{Ind},\ v\text{: } \mathtt{Val}\ (\ \mathtt{add}(\mathtt{add}(t,\ i,\ v),\ i',\ v) = \mathtt{add}(\mathtt{add}(t,\ i',\ v),\ i,\ v)\ )$$

1.4. Combining Traits

`Table` contains a number of totally unconstrained operators (e.g., +). Such traits are not very useful. Additional assertions dealing with these operators could be added to `Table`. However, for modularity, it is often better to include a separate trait by reference. This makes it easier to reuse pieces of other specifications and handbooks. We might add to trait `Table`:

includes `Cardinal`

The theory associated with the including trait is the theory associated with the union of all of the **introduces** and **asserts** clauses of the trait body and the included traits.

It is often convenient to combine several traits dealing with different aspects of the same operator. This is common when specifying something that is not easily thought of as an abstract data type. Consider, for example, the following specifications of properties of relations:

Reflexive: **trait**
  introduces $\_\_ \diamond \_\_$: T, T → Bool
  asserts ∀ *t*: T
    $t \diamond t$

Symmetric: **trait**
  introduces $\_\_ \diamond \_\_$: T, T → Bool
  asserts ∀ *t, t'*: T
    $t \diamond t' == t' \diamond t$

Transitive: **trait**
  introduces $\_\_ \diamond \_\_$: T, T → Bool
  asserts ∀ *t, t', t"*: T
    $(t \diamond t' \wedge t' \diamond t") \Rightarrow t \diamond t"$

Equivalence1: **trait**
  includes Reflexive, Symmetric, Transitive

The trait Equivalence1 has the same associated theory as the following less structured trait:

Equivalence2: **trait**
  introduces $\_\_ \diamond \_\_$: T, T → Bool
  asserts ∀ *t, t', t"*: T
    $t \diamond t$
    $t \diamond t' == t' \diamond t$
    $(t \diamond t' \wedge t' \diamond t") \Rightarrow t \diamond t"$

1.5. Renaming

Equivalence1 relies heavily on the use of the same operator symbol, $\diamond$, and the same sort identifier, T, in three included traits. In the absence of such happy coincidences, renaming can be used to make names coincide, to keep them from coinciding, or simply to replace them with more suitable names, for example,

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

The phrase Tr(name1 **for** name2) stands for the trait Tr with every occurrence of name2 (which must be either a sort or operator name) replaced by name1. If name2 is a sort identifier, this renaming may change the signatures associated with some of the operators in Tr.

If Table were augmented by the **generated by**, **partitioned by**, and **includes** clauses of the two previous sections, the specification

```
SparseArray: trait
  includes Integer,
    Table(Arr for Tab, defined for ∈, assign for add, __[__] for lookup, Int for Ind)
```

would be equivalent to

```
SparseArray: trait
  includes Integer, Cardinal
  introduces
    new: → Arr
    assign: Arr, Int, Val → Arr
    defined: Int, Arr → Bool
    __[__]: Arr, Int → Val
    isEmpty: Arr → Bool
    size: Arr → Card
  asserts
    Arr generated by new, assign
    Arr partitioned by defined, __[__]
    ∀ i, i': Int, val: Val, t: Arr
      assign(t, i, val)[i'] == if i = i' then val else t[i']
      ¬defined(i, new)
      defined(i, assign(t, i', val)) == i = i' ∨ defined(i, t)
      size(new) == 0
      size(assign(t, i, val)) == if defined(i, t) then size(t) else size(t) + 1
      isEmpty(t) == size(t) = 0
```

Note that the infix operator symbol __∈__ was replaced by the operator defined, and that the operator lookup was replaced by the mixfix operator symbol __[__]. Renamings preserve the order of operands.

Any sort or operator in a trait can be renamed when that trait is referenced in another trait. Some, however, are more likely to be renamed than others. It is often convenient to single these out so that they can be renamed positionally. For example, if the header for the `SparseArray` trait had been "`SparseArray(Val):` **trait**", the phrases "**includes** `SparseArray(Int)`" and "**includes** `SparseArray(Int` **for** `Val)`" would be equivalent.

1.6. Stating Intended Consequences

It is not possible to prove the "correctness" of a specification, because there is no absolute standard against which to judge correctness. But specifications can contain errors, and specifiers need help in locating them. Since LSL specifications cannot generally be executed, they cannot be tested in the way that programs are commonly tested. LSL sacrifices executability in favor of brevity, clarity, and flexibility, and provides other ways to check specifications.

This section briefly describes ways in which specifications can be augmented with redundant information to be checked during validation. Chapter 3 defines the checks rigorously. A separate paper discusses the use of LP, the Larch Prover [Garland, Guttag, and Horning 1990] to assist in specification debugging.

Checkable properties of LSL specifications fall into three categories: *consistency*, *theory containment*, and *completeness*. As discussed in Section 1.2, the requirement of consistency makes any trait whose theory contains `true` == `false` illegal.

Claims about theory containment are made using **implies**. Consider the claim that `SparseArray` guarantees that an array with a defined element isn't empty. To indicate that this claim should be checked, we could add to `SparseArray`

implies $\forall$ *a:* `Arr`, *i:* `Int`
  `defined(`*i*, *a*`)` $\Rightarrow$ $\neg$`isEmpty(`*a*`)`

The theory claimed to be implied can be specified using the full power of the language, including equations, **generated by** and **partitioned by** clauses, and references to other traits. In addition to assisting in error detection, implications help readers confirm their understanding, and can simplify reasoning about higher-level traits.

The initial design of LSL incorporated a built-in requirement of completeness. However, we quickly concluded that this was better left to the specifier's discretion. It is useful to check certain aspects of completeness long before a specification is finished, yet most finished specifications (intentionally) don't fully define all their operators. Claims about how complete a specification is are made using **converts**. Adding the claim "**implies converts** `isEmpty`" to `Table` says that the trait's axioms fully define `isEmpty`. This means that, if the interpretations of all the other operators are fixed, there is a unique interpretation of `isEmpty` satisfying the axioms.

Now consider adding the stronger claim "**implies converts** `isEmpty`, `lookup`" to `Table`. The meaning of terms of the form `lookup(new,` i`)` is not defined by the trait, so it isn't possible to verify this claim. The incompleteness could be resolved by adding another axiom to the trait, for example, "`lookup(new,` i`) == errorVal`". However, the specifier of `Table` should not be concerned with whether `Val` has an `errorVal` operator, and should not be required to introduce irrelevant constraints on `lookup`. Extra axioms give readers more details to assimilate. They may preclude useful specializations of a general specification. And sometimes there is no reasonable axiom that would make an operator convertible (consider division by 0).

LSL provides an **exempting** clause that lists terms that need not be defined. The claim "**implies converts** `isEmpty`, `lookup` **exempting** ∀ i: `Ind lookup(new,` i`)`" means that, if interpretations of the other operators and of all terms matching `lookup(new,` i`)` are fixed, there are unique interpretations of `isEmpty` and `lookup` that satisfy the trait's axioms. This is provable from the specification.

1.7. Recording Assumptions

It is useful to construct general specifications that can be specialized in a variety of ways. Consider, for example,

```
Bag(E): trait
  introduces
    { }: → B
    insert, delete: E, B → B
    __∈__: E, B → Bool
  asserts
    B generated by { }, insert
    B partitioned by delete, ∈
    ∀ b: B, e, e': E
      ¬(e ∈ { })
      e ∈ insert(e', b) == e = e' ∨ e ∈ b
      delete(e, { }) == { }
      delete(e', insert(e, b)) == if e = e' then b else insert(e, delete(e', b))
```

We might specialize this to `IntegerBag` by renaming `E` to `Int` and including it in a trait in which operators dealing with `Int` are specified, for example,

```
IntegerBag: trait
  includes Integer, Bag(Int)
```

The interactions between `Integer` and `Bag` are very limited. Nothing in `Bag` makes any assumptions about the meaning of the operators, such as `0`, $+$, and $<$, that are defined in `Integer`. Consider, however, extending `Bag` to `Bag1` by adding an operator `rangeCount`,

```
Bag1(E): trait
  includes Bag, Cardinal
  introduces
    rangeCount: E, E, B → Card
    __<__: E, E → Bool
  asserts ∀ e, e', e": E, b: B
    rangeCount(e, e', { }) == 0
    rangeCount(e, e', insert(e", b)) ==
          rangeCount(e, e', b) + ( if e < e" ∧ e" < e' then 1 else 0 )
```

As written, `Bag1` makes no assumptions about the properties of the $<$ operator. Suppose, however, that we wish to require that, in any specialization of this trait, $<$ provides an ordering on the values of sort `E`. We can add such a requirement with an *assumption*:

```
Bag2(E): trait
  assumes TotalOrder(E)
  includes Bag, Cardinal
  introduces rangeCount: E, E, B → Card
  asserts ∀ e, e', e": E, b: B
    rangeCount(e, e', { }) == 0
    rangeCount(e, e', insert(e", b)) ==
          rangeCount(e, e', b) + ( if e < e" ∧ e" < e' then 1 else 0 )
  implies ∀ e, e', e": E, b: B
    e' ≤ e" ⇒ rangeCount(e, e', b) ≤ rangeCount(e, e", b)
```

The theory associated with `Bag2` is the same as if `TotalOrder(E)` had been included rather than assumed; `Bag2` inherits all the declarations and axioms of `TotalOrder`. Therefore, the assumption can be used to derive various properties of `Bag2`, including the implication that `rangeCount` is monotonic in its second argument.

The difference between **assumes** and **includes** appears when `Bag2` is used in another trait. Whenever a trait with assumptions is included or assumed, its assumptions must be *discharged*. For example, in

```
IntegerBag2: trait
  includes Integer, Bag2(Int)
```

the assumption to be discharged is that the (renamed) theory associated with `TotalOrder` is a subset of the theory associated with `Integer`. When a trait includes a trait with assumptions, it is often possible to determine that these assumptions are discharged by noticing that the same traits are assumed or included in the including trait. For example, `Integer` itself might directly include `TotalOrder`.

1.8. Built-In Operators and Operator Overloading

In our examples, we have freely used various Boolean operators, plus some heavily overloaded and apparently unconstrained operators: **if__then__else__**, =, and ≠. Although these operators are definable within LSL, they are built into the language. This allows them to have appropriate syntactic precedence. More importantly, it guarantees that they have consistent meanings in all LSL specifications, so readers can rely on their intuitions about them. For example, the built-in definition of = guarantees that for any terms t1 and t2, t1 = t2 == `true` if and only if t1 == t2.

In addition to the built-in overloaded operators, LSL provides for user-defined overloadings. Each operator must be declared in an **introduces** clause and consists of an identifier (e.g., `empty`) or operator symbol (e.g., __<__) and a signature. The signatures of most occurrences of overloaded operators are deducible from context. Consider, for example,

```
OrderedString(E, Str): trait
  assumes TotalOrder(E)
  introduces
    empty: → Str
    insert: E, Str → Str
    __<__: Str, Str → Bool
  asserts
    Str generated by empty, insert
    ∀ e, e': E, s, s': Str
      empty < insert(e, s)
      ¬(s < empty)
      insert(e, s) < insert(e', s') == e < e' ∨ (e = e' ∧ s < s')
  implies TotalOrder(Str)
```

The operator symbol < is used in the last equation to denote two different operators, one relating terms of sort `Str` and the other, terms of sort `E`, but their contexts determine unambiguously which is which. LSL provides notations for disambiguating an overloaded operator if context does not suffice. Any subterm of a term can be qualified by its sort. For example, "`a:S` = `b`" explicitly indicates that `a` is of sort `S`. Since the two operands of = must have the same sort, this qualification also implicitly defines the signatures of = and `b`. Outside of terms, overloaded operators can be disambiguated by directly affixing their signatures.

1.9. Enumerations, Tuples, and Unions

Enumerations, tuples, and unions provide compact, readable representations for common kinds of theories. They are just syntactic shorthands for things that could be written in LSL without them.

The enumeration shorthand defines a finite set of distinct constants and an operator that enumerates them. For example,

`Temp` **enumeration of** `cold`, `warm`, `hot`

is equivalent to including a trait whose body is:

```
introduces
  cold, warm, hot: → Temp
  succ: Temp → Temp
asserts
  Temp generated by cold, warm, hot
  equations
    cold ≠ warm
    cold ≠ hot
    warm ≠ hot
    succ(cold) == warm
    succ(warm) == hot
```

The tuple shorthand is used to introduce fixed-length tuples. For example,

`C` **tuple of** `hd: E, tl: S`

is equivalent to including a trait whose body is:

```
introduces
  [__, __]: E, S → C
  __.hd: C → E
  __.tl: C → S
  set_hd: C, E → C
  set_tl: C, S → C
asserts
  C generated by [__, __]
  C partitioned by .hd, .tl
  ∀ e, e': E, s, s': S
    [e, s].hd == e
    [e, s].tl == s
    set_hd([e, s], e') == [e', s]
    set_tl([e, s], s') == [e, s']
```

Each field name (e.g., hd) is incorporated in two distinct operators (e.g., __.hd:C→E and set_hd:C,E→C).

The union shorthand corresponds to the tagged unions found in many programming languages. For example,

```
S union of atom: A, cell: C
```

is equivalent to including a trait whose body is:

```
S_tag enumeration of atom, cell
introduces
  atom: A → S
  cell: C → S
  __.atom: S → A
  __.cell: S → C
  tag: S → S_tag
asserts
  S generated by atom, cell
  S partitioned by .atom, .cell, tag
  ∀ a: A, c: C
    atom(a).atom == a
    cell(c).cell == c
    tag(atom(a)) == atom
    tag(cell(c)) == cell
```

Each field name (e.g., atom) is incorporated in three distinct operators (e.g., atom:→S_tag, atom:A→S, and __.atom:S→A).

1.10. Characters and symbols

LSL was designed for use with an open-ended collection of programming languages, support tools, and input/output facilities, each of which may have its own lexical conventions and capabilities. To avoid conflicts, LSL assigns fixed meanings to only a small number of characters. To conform to local conventions and to exploit locally available capabilities, LSL's character and token classes are open-ended, and can be tailored for particular uses by *initialization files*, as discussed in Appendix II.

Contiguous sequences of identifier characters (alphanumerics and underscore) and contiguous sequences of operator characters (asterisk, plus, minus, period, slash, less-than, equal, greater-than) form single tokens. Whitespace characters are insignificant except for separating tokens. Each of the remaining characters constitutes a separate token.

There are several semantically equivalent forms of LSL. Any of these forms can be mechanically translated into any other without losing information.

- *Presentation forms* are used in environments with rich sets of characters (e.g., $\forall, \wedge, \vee, \in$), including this report.
- *Interchange form* is an encoding of LSL using a subset of the ASCII character set. Characters outside this subset are represented by *extended characters*—sequences of characters from the subset, set off by a backslash (or another designated character). Interchange form is the "lowest common denominator" for LSL. Each Larch tool must be able to parse it, and to generate it on demand.
- *Interactive forms* are used by Larch editors, browsers, checkers, etc., for input and output. Many will not be limited to character strings for input and output, and some may impose additional constraints and equivalences (e.g., case folding, operator precedence).

1.11. Further Examples

We have now covered all the facilities of the Larch Shared Language. The next series of examples illustrates their coordinated use.

The trait `Container` abstracts the common properties of data structures that contain elements, such as sets, bags, queues, stacks, and strings. `Container` is useful both as a starting point for specifications of many different data structures and as an assumption when defining generic operators over such data structures.

The **generated by** clause in `Container` asserts that each value of sort `C` can be constructed from `new` by repeated applications of `insert`. This assertion is carried along when Container is included in or assumed by other traits, even if they introduce additional operators with range `C`. Theorems proved by induction over `new` and `insert` will be valid in the theories associated with all such traits.

```
Container(E, C): trait
  introduces
    new: → C
    insert: E, C → C
  asserts C generated by new, insert
```

The trait `LinearContainer` includes `Container`. It constrains `new` and `insert`, inherited from `Container`, as well as the additional operators it introduces. The **partitioned by** clause indicates that `next`, `rest`, and `isEmpty` form a complete set of observers for sort `C`: for any terms `t1` and `t2` of sort `C`, if the equalities `next(t1) == next(t2)`, `rest(t1) == rest(t2)`, and `isEmpty(t1) == isEmpty(t2)` all hold, then `t1 == t2`. The axioms for `next` and `rest` are intentionally very weak (defining their meaning only for single-element containers) so that `LinearContainer` can be specialized to define stacks, queues, priority queues, and strings. The **converts** clause adds checkable redundancy to the specification by claiming that this trait fully defines `isEmpty`.

```
LinearContainer(E, C): trait
  includes Container
  introduces
    isEmpty: C → Bool
    next: C → E
    rest: C → C
  asserts
    C partitioned by next, rest, isEmpty
    ∀ c: C, e: E
      isEmpty(new)
      ¬isEmpty(insert(e, c))
      next(insert(e, new)) == e
      rest(insert(e, new)) == new
  implies converts isEmpty
```

`PriorityQueue` specializes `LinearContainer` by adding another operator, ∈, and by further constraining `next`, `rest`, and `insert`. The first implication states a fact that can be proved using the induction rule inherited from `Container`. It may be helpful in reasoning about `PriorityQueue` and may help readers solidify their understanding of the trait. The second implication states that the trait defines `next` and `rest` (except when applied to `new`), `isEmpty`, and ∈. The axioms that convert `isEmpty` are inherited from `LinearContainer`.

```
PriorityQueue(E, Q): trait
  assumes TotalOrder(E)
  includes LinearContainer(Q for C)
  introduces __∈__: E, Q: → Bool
  asserts ∀ e, e': E, q: Q
    next(insert(e, q)) ==
      if q = new then e else if next(q) < e then next(q) else e
    rest(insert(e, q)) ==
      if q = new then new else if next(q) < e then insert(e, rest(q)) else q
    ¬(e ∈ new)
    e ∈ insert(e', q) == e = e' ∨ e ∈ q
  implies
      ∀ q: Q, e: E
        e ∈ q ⇒ ¬(e < next(q))
      converts next, rest, isEmpty, ∈ exempting next(new), rest(new)
```

Unlike the preceding traits in this section, `PriorityQueue` specifies an abstract data type constructor. In such a trait there is a distinguished sort, sometimes called the "type of interest" [Guttag 1975] or "data sort" [Burstall and Goguen 1980]. An abstract data type's operators can be categorized as generators, observers, and extensions (sometimes in more than one way). A set of generators produces all the values of the distinguished sort. The extensions are the remaining operators whose range is the distinguished sort. The observers are the operators whose domain includes the distinguished sort and whose range is some other sort. An abstract data type specification usually converts the observers and extensions. The distinguished sort is usually partitioned by at least one subset of the observers and extensions. For example, in `PriorityQueue`, `Q` is the distinguished sort, `new` and `insert` form a generator set, `rest` is an extension, `next`, `isEmpty`, and $\in$ are the observers, and `next`, `rest`, and `isEmpty` form a partitioning set.

A good heuristic for generating enough equations to adequately define an abstract data type is to write an equation defining the result of applying each observer or extension to each generator [Guttag 1975]. For `PriorityQueue`, this rule suggests writing equations for `rest(new)`, `next(new)`, `isEmpty(new)`, $e \in$ `new`, `rest(insert(`e, q`))`, `next(insert(`e, q`))`, `isEmpty(insert(`e, q`))`, and $e \in$ `insert(`e', q`)`. `PriorityQueue` contains explicit equations for four of the eight, and inherits equations for two more from `LinearContainer`. The remaining two terms, `next(new)` and `rest(new)`, are explicitly exempted.

The next two traits, `PairwiseExtension` and `PairwiseSum`, specify generic operators that can be used with various kinds of ordered containers.

Given a binary operator on elements, $\circ$, PairwiseExtension defines a new binary operator on containers, $\odot$. The result of applying $\odot$ to a pair of containers is a container whose elements are the results of applying ∘ to corresponding pairs of their elements. The assumption of `LinearContainer` ensures that the notion of "corresponding pair" is well-defined; to understand why `Container` would not suffice, imagine defining ⊙ consistently for a `Bag`. The **exempting** clause indicates that, although the result of applying ⊙ to containers of unequal size is not specified, this is not an oversight. Since ∘ is totally unconstrained in this trait, there aren't yet many interesting implications to state.

```
PairwiseExtension(E, C): trait
  assumes LinearContainer
  introduces
    __∘__: E, E → E
    __⊙__: C, C → C
  asserts ∀ e, e': E, c, c': C
    new ⊙ new == new
    insert(e, c) ⊙ insert(e', c') == insert(e ∘ e', c ⊙ c')
  implies converts ⊙
    exempting ∀ e: E, c: C
      new ⊙ insert(e, c),
      insert(e, c) ⊙ new
```

Now we specialize `PairwiseExtension` by binding ∘ to an operator, +, whose definition is to be taken from the trait `Cardinal`.

```
PairwiseSum(C): trait
  assumes LinearContainer(Card for E)
  includes Cardinal,
    PairwiseExtension(Card for E, + for ∘, ⊕ for ⊙)
  implies (Associative, Commutative) (⊕ for ∘, C for T)
```

The validity of the implication that ⊕ is associative and commutative stems from the replacement of ∘ by +, whose axioms in a suitable trait `Cardinal` would imply its associativity and commutativity. The implication could then be proved by induction over `new` and `insert`.

1.12. Significant Decisions in the Design of LSL

Our basic assumption was that specifications will be constructed and checked incrementally. This led us to a design that ensures that adding axioms to a trait never invalidates theorems. The need to maintain this monotonicity property led us to construe the equations of a trait as denoting a first-order theory. Neither the initial algebra nor the final algebra interpretation of a set of equations has this property.

Many traits correspond to complete abstract data types, but many others do not. So we included independent constructs to identify complete sets of constructors (**generated by**) and complete partitioning sets (**partitioned by**). Separating them provides useful flexibility.

The freedom to rename any of a trait's operators or sorts is also useful. In effect, all names appearing in a trait are formal parameters. An early version of LSL had only explicit lambda abstraction. We soon discovered that it was hard to get a trait's formal parameter list "right." If we kept it short, we often wished to substitute for a name that hadn't been included. If we used a longer list, we frequently didn't need to rename most of the potential parameters, and supplied the same names for the actuals as the formals. This experience led us to abolish explicit parameter lists in LSL 1.1 [Guttag and Horning 1986]; all renaming was of the form "id1 **for** id2." But the restriction to explicit renaming also proved cumbersome. In the current design, the specifier can choose to rename either positionally or explicitly.

Specifiers shouldn't start from scratch each time; LSL specifications are reusable. Handbooks of LSL specifications—some specialized for particular application domains—play an important role in specification development. (The examples used in this report are, for expository purposes, atypically complete.) We chose not to build into LSL many constructs that can easily be supplied by handbook traits.

Reading specifications is an important activity. People read syntactic objects (traits), rather than semantic objects (theories). So we chose to define the mechanisms for combining LSL specifications syntactically. However, for each of our combining operations on traits, there is a corresponding operation on theories such that the theory associated with any combination of traits is the same as the combination of their associated theories.

There is a tension in the design of the syntax for terms. On one hand, we want to allow specifiers as much notational flexibility as we can. On the other, it is important that both people and tools be able to parse terms in interface language specifications without reference to operator declarations (which are off in LSL traits). Our grammar for terms is fairly flexible, but—because there is no way to specify the precedence of user-defined operators—requires more parentheses than we would like.

Operator names in LSL include full signatures, unlike many programming languages, where overloaded operators are qualified by a single type or by a module name. This decision resulted from our desire to make heavy use of overloading in interface specifications. Contextual disambiguation means that it is not usually necessary to clutter up terms with explicit sorts.

We made a conscious attempt to reduce the number of characters reserved by LSL, to avoid conflicts with programming language usages (which will be reflected in interface languages), to avoid conflicts with notations from mathematics and application domains (which will be reflected in handbooks), and to avoid problems with different character sets in different environments. There isn't any real choice about commas, colons, and parentheses; fortunately, their uses in mathematics and most programming languages are compatible. We reserved these four characters and then used them throughout, in preference to other characters, such as semicolons and brackets. We took almost exactly the opposite approach for keywords, which appear in traits, but not in interface specifications. We deliberately chose distinctive keywords and reserved them.

LSL's constructs for introducing checkable redundancy into specifications were chosen to expose classes of errors that we expect to be common. These facilities help specifiers increase the chance that a specification with an unintended meaning will be detectably illegal, in much the same way that type systems increase the chance that an erroneous program will be detectably illegal. In contrast to our emphasis on syntactic mechanisms for combining traits, we included a number of semantic constraints on their legality. This means that a theorem prover is needed to fully check traits [Garland, Guttag, and Horning 1990]. The constructs for checking have other costs: LSL would be considerably smaller without them, and it takes about as long to learn the part of the language involved with checking as it does to learn the part required to generate theories.

The Larch approach frequently leads to traits in which many things are left unconstrained, so traits are not required to completely define all operators. Instead, **converts** clauses allow the specifier to include checkable claims about completeness, which can reflect the trait's intended uses in interface specifications. Exactly what it means to completely define an operator was a delicate design issue for LSL. The meaning of a **converts** clause is that, given any fixed interpretations for the other operators and the exempted terms, the interpretations of the converted operators that satisfy the trait's axioms are unique.

LSL 1.1 contained two additional constructs, **imports** and **constrains**, that were used to claim that one theory was a conservative extension of another. We found that these constructs were difficult to explain, to use effectively, and to check, so we have dropped them from the language.

In many respects, LSL is distinguished from other specification languages as much by what it doesn't include as by what it does.

LSL provides no construct for hiding operators. The hiding constructs of other specification languages [e.g., Burstall and Goguen 1980] allow the introduction of auxiliary operators that don't have to be implemented. These operators are not completely hidden, since they must be read to understand the specification, and they are likely to appear in reasoning based on the specification. The two-tiered structure of Larch specifications means that none of the operators appearing in an LSL trait have to be implemented; they are all auxiliary functions to be used in writing interface specifications. We could say that the entire LSL tier is "hidden."

LSL does not provide constructs for specifying partial functions or error algebras. There is no mechanism other than sort checking for restricting the domain of operators. Terms such as `lookup(new`, i`)` are allowed, and no special error elements are built into the language to represent the values of such terms. As discussed in [Guttag, Horning, and Wing 1985a], preconditions and errors are handled in Larch interface languages.

Similarly, nondeterminism is left to the interface languages. It is frequently useful to write incomplete specifications that allow different interpretations of equality (and have non-isomorphic models). Thus, for many traits there are terms that are neither provably equal nor provably unequal. However, it is always the case in LSL that for every term t, t == t. The mathematical basis of algebra, and of LSL, depends on the validity of freely substituting equals for equals. This would be destroyed by the introduction of "nondeterministic functions."

We chose not to include higher-order entities in LSL. Traits are simple textual objects. Their associated theories are first-order theories. We sidestepped the subtle semantic problems associated with parameterized theories, theory parameters, and the like [Ehrig and Mahr 1985]. **Includes** and **assumes** clauses, together with renamings, make possible much of the reuse for which higher-order theories are advocated.

1.13. Grammar

```
trait          ::= simpleId [ ( { name [ : signature ] }+, ) ] : trait
                   { shorthand | external }* opPart* propPart* [ consequences ]
name           ::= simpleId | opForm
opForm         ::= if __ then __ else __
                   | [ __ ] { simpleOp | logicalOp | eqOp } [ __ ]
                   | [ __ ] openSym [ placeList ] closeSym [ __ ]
                   | [ __ ] . simpleId
placeList      ::= __ { { sepSym | , } __ }*
signature      ::= sort*, → sort
sort           ::= simpleId
shorthand      ::= enumeration | tuple | union
enumeration    ::= sort enumeration of simpleId+,
tuple          ::= sort tuple of fields+,
union          ::= sort union of fields+,
fields         ::= simpleId+, : sort
external       ::= { includes | assumes } traitRef+,
traitRef       ::= { simpleId | ( simpleId+, ) } [ ( renaming ) ]
renaming       ::= replace+, | name+, { , replace }*
replace        ::= name for name [ : signature ]
opPart         ::= introduces opDcl+
opDcl          ::= name+, : signature
propPart       ::= asserts genPartition* eqPart
genPartition   ::= sort { generated | partitioned } by operator+,
operator       ::= name [ : signature ]
eqPart         ::= [ equations eqSeq ] { ∀ varDcl+, eqSeq }*
varDcl         ::= simpleId+, : sort
eqSeq          ::= equation { eqSepSym equation }*
equation       ::= term [ == term ]
term           ::= logicalTerm | if term then term else term
logicalTerm    ::= equalityTerm { logicalOp equalityTerm }*
equalityTerm   ::= simpleOpTerm [ eqOp simpleOpTerm ]
simpleOpTerm   ::= simpleOp+ secondary
                   | secondary simpleOp+
                   | secondary { simpleOp secondary }*
secondary      ::= primary | [ primary ] bracketed [ : sort ] [ primary ]
bracketed      ::= openSym [ term { { sepSym | , } term }* ] closeSym
primary        ::= { ( term ) | simpleId [ ( term+, ) ] } { . simpleId | : sort }*
consequences   ::= implies { traitRef*, genPartition* eqPart
                             | [ traitRef+, genPartition+ ] eqSeq } conversion*
conversion     ::= converts operator+, [ exempting [ ∀ varDcl+, ] term+, ]
```

Chapter 2

Language Definition

This chapter is a self-contained definition of the Larch Shared Language, Version 2.3. It defines the syntax and static semantics of LSL and the theory associated with each LSL specification.

- Section 1 defines the semantic core language (SCL), a small language (similar to a subset of LSL) that is sufficient to express any theory expressible in LSL. The semantics of LSL is defined by giving its translation into SCL.
- Section 2 defines a simple, unstructured subset of LSL and its translation into SCL.
- Sections 3–12 define successive language extensions. They extend the grammar, describe additional checking, and provide a normalization of each extension into the previously defined subset. Normalized specifications are further subject to the checking defined for the target subset. The theory associated with a specification is the theory associated with the translation into SCL of its normalization.
  - Section 3 introduces structural facilities for combining specifications.
  - Sections 4–5 introduce facilities for adding redundancy to a specification by stating intended consequences.
  - Sections 6–12 introduce syntactic amenities.
- The Appendices discuss details of the logic used for LSL theories, the lexical structure of the language, and the grammatical notation used in this report.

2.1. SCL: The Semantic Core Language

Grammar

presentation ::= { *generators* | *partitions* | *equation* }*

generators ::= *sort* **generated by** *operator*⁺**,**

partitions ::= *sort* **partitioned by** *operator*⁺**,**

operator ::= *name* **:** *signature*

signature ::= *domain* → *range*

domain ::= *sort****,**

range ::= *sort*

sort ::= *simpleId*

equation ::= *expression* == *expression*

expression ::= *operator* [**(** *expression*⁺**,)**] | *variable*

variable ::= *simpleId* **::** *sort*

Definitions

- A *presentation* is *syntactically legal* if it satisfies the context-free grammar and the context-sensitive checks.
- The *sort* of an *expression* of the form *operator* [(*expression*$^+$**,**)] is *operator*'s *range*, and the sort of an *expression* of the form *simpleId*::*sort* is *sort*.
- A *constant* is an operator with an empty *domain*.

Context-sensitive checking

- The *range* of each *operator* in a *generators* must be the *sort* of the *generators*.
- At least one *operator* in a *generators* must have a *domain* in which the *sort* of the *generators* does not occur.
- The *domain* of each *operator* in a *partitions* must include the *sort* of the *partitions*.
- The *range* of at least one *operator* in a *partitions* must be different from the *sort* of the *partitions*.
- In each *equation*, the sorts of the two *expressions* must be the same.
- In each *expression* of the form *operator* [(*expression****,**)] , the *operator*'s *domain* must be the sequence of the sorts of the *expressions*.

Associated Theory

With each *presentation*, we associate a theory in typed first-order logic with equality.[2] Theories are constructed using the alphabet of SCL symbols for *sorts*, *variables*, and *operators*. We identify the SCL symbols ==, `true`:→`Bool`, and `false`:→`Bool` with the logical symbols =, true, and false, respectively.

The theory associated with a *presentation* is the smallest theory containing the set of formulas constructed as follows:

- The theory contains the universal closure of each *equation*.
- For each *generators*, S **generated by** $\text{op}_1, \ldots, \text{op}_n$, and for each formula P and each variable y of sort S, the theory contains the universal closure of the induction formula

$$(\forall y P) \equiv \bigwedge_{1 \leq i \leq n} \forall x_{i,1} \ldots \forall x_{i,k_i} \left(\left[\bigwedge_{j:sort(x_{i,j})=\text{S}} P[y \leftarrow x_{i,j}] \right] \Rightarrow P[y \leftarrow t_i] \right)$$

[2] Appendix I contains some relevant definitions and examples.

where t_i is the expression $\mathrm{op}_i(x_{i,1}, \ldots, x_{i,k_i})$ and the $x_{i,j}$ are distinct variables of the appropriate sorts that do not appear in P.

- For each *partitions*, S **partitioned by** $\mathrm{op}_1, \ldots, \mathrm{op}_n$, and for each pair of variables y and z of sort S, the theory contains the universal closure of the formula

$$(y = z) \equiv \bigwedge_{1 \le i \le n} \forall x_{i,1} \ldots \forall x_{i,k_i} \left(\bigwedge_{j:sort(x_{i,j}) = \mathrm{S}} t_i[x_{i,j} \leftarrow y] = t_i[x_{i,j} \leftarrow z] \right)$$

where t_i is the expression $\mathrm{op}_i(x_{i,1}, \ldots, x_{i,k_i})$ and the $x_{i,j}$ are distinct variables of the appropriate sorts that are distinct from y and z.

2.2. Simple Traits

Grammar

trait ::= *simpleId* **: trait** *traitBody*
traitBody ::= *simpleTrait*
simpleTrait ::= *opPart** *propPart**
opPart ::= **introduces** *opDcl*+
opDcl ::= *name*+**,** **:** *signature*
name ::= *simpleId* | *opForm*
opForm ::= **if** __ **then** __ **else** __
| [__] { *simpleOp* | *logicalOp* | *eqOp* } [__]
| [__] *openSym* [*placeList*] *closeSym* [__]
| [__] **.** *simpleId*
placeList ::= __ { { *sepSym* | **,** } __ }*
signature ::= *domain* → *range*
domain ::= *sort****,**
range ::= *sort*
sort ::= *simpleId*
propPart ::= **asserts** *props*
props ::= { *generators* | *partitions* }* *eqPart*
generators ::= *sort* **generated by** *operator*+**,**
partitions ::= *sort* **partitioned by** *operator*+**,**
operator ::= *name* **:** *signature*
eqPart ::= [**equations** *eqSeq*] { *quantifier* *eqSeq* }*
quantifier ::= ∀ *varDcl*+**,**
varDcl ::= *simpleId*+**,** **:** *sort*
eqSeq ::= *equation* { *eqSepSym* *equation* }*
equation ::= *term* == *term*
term ::= *name* [**(** *term*+**,** **)**] **:** *sort*

The definition of *term* is replaced, not extended, in Section 2.8. The "subsets" of Sections 2.2–7 allow non-LSL *terms* that are useful in the translation of full to SCL.

Definitions

- A *trait*'s *theory* is the theory associated with its translation into SCL.
- A *trait* or *traitBody* is *syntactically legal* if it satisfies the context-free grammar and the context-sensitive checks and its translation into SCL is syntactically legal.
- A *trait* or *traitBody* is *semantically legal* if it is syntactically legal and satisfies the semantic checks.
- The *operator list* of an *opDcl* $\text{op}_1, \ldots, \text{op}_n$: sig is op_1: sig ... op_n: sig .
- The *operator list* of a *simpleTrait* is **introduces** followed by the union of the operator lists of its *opDcls*.[3]
- The *variable list* of a *varDcl* $\text{v}_1, \ldots, \text{v}_n$: S is v_1: S, ..., v_n: S .
- The *variable list* of an *eqPart* is ∀ followed by the union of the variable lists of its *varDcls*.
- op:S and op:→S are *occurrences* of the constant *operator* op:→S .
- $\text{op}(\text{t}_1{:}\text{S}_1, \ldots, \text{t}_n{:}\text{S}_n){:}\text{S}$ and $\text{op}{:}\text{S}_1, \ldots, \text{S}_n \rightarrow \text{S}$ are *occurrences* of the *operator* $\text{op}{:}\text{S}_1, \ldots, \text{S}_n \rightarrow \text{S}$.

Context-sensitive checking

- No *simpleId* may occur more than once in any *quantifier*.
- If id:→S is in the operator list of a *simpleTrait*, then id:S may not be in the variable list of any of its *eqParts*.
- Each *operator* in the translation of a *simpleTrait* must be in its operator list.
- Each *variable* appearing in the translation of an *eqPart* must be in its variable list.

Translation

A *trait* is translated to a *presentation* in SCL by retaining its *generators* and *partitions*, deleting its *opParts*, and translating each *propPart* by deleting its *quantifier* and translating each *term* to an *expression* by replacing

- Each *term* of the form id:S by the constant *operator* id:→S if id:S is in the operator list of the containing *eqPart*, and by the *variable* id::S otherwise.
- Each *term* of the form $\text{op}(\text{t}_1{:}\text{S}_1, \ldots, \text{t}_n{:}\text{S}_n){:}\text{S}$ by the *expression* $\text{op}{:}\text{S}_1, \ldots, \text{S}_n \rightarrow \text{S}(\text{e}_1, \ldots, \text{e}_n)$, where $\text{e}_1, \ldots, \text{e}_n$ are the translations of $\text{t}_1{:}\text{S}_1, \ldots, \text{t}_n{:}\text{S}_n$, respectively.

[3] For convenience, we will speak of the concatenation of lists as their "union."

Semantic checking

- Each *trait* must be *consistent:* the theory associated with its translation must not contain the formula "true = false".

2.3. Externals

Add to the grammar the productions:

traitBody	::=	*traitContext*$^+$ *simpleTrait*
traitContext	::=	*external*
external	::=	*includes* | *assumes*
includes	::=	**includes** *traitRef*$^+$**,**
assumes	::=	**assumes** *traitRef*$^+$**,**
traitRef	::=	{ *simpleId* | **(** *simpleId*$^+$**,** **)** } [**(** *renaming* **)**]
renaming	::=	{ *sortReplace* | *opReplace* }***,**
sortReplace	::=	*newSort* **for** *oldSort*
newSort	::=	*sort*
oldSort	::=	*sort*
opReplace	::=	*newOp* **for** *oldOp*
newOp	::=	*name*
oldOp	::=	*operator*

Definitions

- The *name mapping* associated with a *renaming* is defined as follows:
  - Simultaneously, for each *opReplace*, replace the *name* part of each occurrence of its *oldOp* by its *newOp*.
  - Then, simultaneously, for each *sortReplace*, replace each occurrence of its *oldSort* by its *newSort*.
- The *normalization* of a *traitRef* is the image, under its name mapping, of the union of the normalizations of the referenced *traits*.
- The *operator list* of a *trait* is the union of the operator list of its *simpleTrait* and the operator lists of the *traitRefs* in its *externals*.
- The *operator list* of a *traitRef* is the image, under its name mapping, of the union of the operator lists of the normalizations of the referenced *traits*.
- The *sort set* of a *trait*, or a *traitRef*, is the set of *sorts* appearing in its operator list.

- The *assertion list* of a *trait* is the union of its *propPart** and the images of the assertion lists of the *traits* referenced in its *includes* under their name mappings.
- The *local assumption list* of a *trait* is the union of the images, under their name mappings, of the local assumption and assertion lists of the *traits* referenced in its *assumes*.
- The *inherited assumption list* of a *trait* is the union of the images, under their name mappings, of the local assumption lists of the *traits* referenced in its *includes*.

Context-sensitive checking

- No *external* may be recursive.
- No *sort* may occur as an *oldSort* more than once in a *renaming*.
- Each *oldSort* must be in the sort set of a *trait* referenced by the enclosing *traitRef*.
- No *operator* may occur as an *oldOp* more than once in a *renaming*.
- Each *oldOp* must be in the operator list of a *trait* referenced by the enclosing *traitRef*.

Semantic checking

- The theory of each *trait* must contain the theory of the *traitBody* consisting of the union of its operator list and its inherited assumption list.

Normalization

- Replace the *traitBody* of each *trait* by the union of its operator list, its assertion list, and its local assumption list.

2.4. Consequences

Add to the grammar the productions:

traitBody	::=	*traitContext** *simpleTrait consequences*
consequences	::=	**implies** *conseqProps*
conseqProps	::=	*traitRef**, *genPartition** *eqPart*
		| [*traitRef*$^+$, *genPartition*$^+$] *eqSeq*

Definition

- The *traitBody* associated with a *consequences* **implies** Refs Props is
  includes Refs opList **asserts** Props
  where opList is the operator list of the enclosing *traitBody*.

Context-sensitive checking

- The *traitBody* associated with the *consequences* must be syntactically legal.

Semantic checking

- The theory of the enclosing *trait* must contain the theory of the *traitBody* associated with the *consequences*.

Normalization

- Remove the *consequences*.

2.5. Converts

Add to the grammar the productions:

consequences	::=	**implies** *conseqProps conversion*$^+$
conversion	::=	**converts** *operator*$^+$**,** [*exemption*]
exemption	::=	**exempting** [*quantifier*] *term*$^+$**,**

Definition

- The *traitBody* associated with a *conversion*

  converts $\mathrm{op}_1, \ldots, \mathrm{op}_n$ **exempting** $\forall$ Vars $\mathrm{t}_1, \ldots, \mathrm{t}_m$

  in trait T is

  includes T (op'_1 **for** $\mathrm{op}_1, \ldots, \mathrm{op}'_n$ **for** op_n), T
  asserts $\forall$ Vars
  $\mathrm{t}'_1 == \mathrm{t}_1$
  $\vdots$
  $\mathrm{t}'_m == \mathrm{t}_m$
  implies
  $\forall\ x_1{:}\ \mathrm{S}_{1,1}, \ldots, x_{k_1}{:}\ \mathrm{S}_{1,k_1}$
  $\mathrm{op}'_1(x_1{:}\ \mathrm{S}_{1,1}, \ldots, x_{k_1}{:}\ \mathrm{S}_{1,k_1}){:}\mathrm{S}_1 == \mathrm{op}_1(x_1{:}\ \mathrm{S}_{1,1}, \ldots, x_{k_1}{:}\ \mathrm{S}_{1,k_1}){:}\mathrm{S}_1$
  $\vdots$
  $\forall\ x_n{:}\ \mathrm{S}_{n,1}, \ldots, x_{k_n}{:}\ \mathrm{S}_{n,k_n}$
  $\mathrm{op}'_n(x_1{:}\ \mathrm{S}_{n,1}, \ldots, x_{k_n}{:}\ \mathrm{S}_{n,k_n}){:}\mathrm{S}_n == \mathrm{op}_n(x_1{:}\ \mathrm{S}_{n,1}, \ldots, x_{k_n}{:}\ \mathrm{S}_{n,k_n}){:}\mathrm{S}_n$

  where

  - $\mathrm{op}'_1, \ldots, \mathrm{op}'_n$ are distinct fresh *names*,
  - $\mathrm{t}'_1, \ldots, \mathrm{t}'_m$ are the *terms* obtained from $\mathrm{t}_1, \ldots, \mathrm{t}_m$ by replacing the *names* in occurrences of each op_i by op'_i, and
  - $\mathrm{S}_{i,1}, \ldots, \mathrm{S}_{i,k_i} \rightarrow \mathrm{S}_i$ is the *signature* of op_i.

Context-sensitive checking

- The *traitBody* associated with each *conversion* must be syntactically legal.
- Each *term* in an *exemption* must contain an occurrence of an *operator* in the enclosing *conversion*.

Semantic checking

- The *traitBody* associated with each *conversion* must be semantically legal.

Normalization

- Remove each *conversion*.

2.6. Positional Renaming

Add to the grammar the productions:

trait	::=	*simpleId* **(** *formalList* **) : trait** *traitBody*
formalList	::=	*formal*$^+$**,**
formal	::=	*sort* | *operator*
renaming	::=	*actual*$^+$**,** { **,** { *sortReplace* | *opReplace* } }*
actual	::=	*newSort* | *newOp*

Context-sensitive checking

- Each *sort* in a *formalList* must be in the sort set of the enclosing *trait*.
- Each *operator* in a *formalList* must be in the operator list of the enclosing *trait*.
- In a *renaming* with *actuals*, the number of *actuals* must equal the number of *formals* in the *formalList* of each referenced *trait*.

Normalization

- Replace each *actual* in a *renaming* by *actual* **for** *formal*, where *formal* is in the corresponding position in the *formalList* of the referenced *trait*.
- Remove each *formalList*.

2.7. Implicit Signatures and Sorts

Add to the grammar the productions:

operator	::=	*name*
term	::=	*name* [(*term*⁺,)]

Definitions

- Any *operator* of the form opName:sig is a *completion* of the *abbreviated operator* opName, unless opName is also in the sort set of the enclosing *trait*.
- Any *term* of the form t:S is a *completion* of the *abbreviated term* t.

Context-sensitive checking

- For each abbreviated *operator* in a *traitRef* there must be a unique completion in the referenced *traits*' operator lists.
- For each abbreviated *operator* in a *formalList* or *conversion* there must be a unique completion in the enclosing *trait*'s operator list.
- For each abbreviated *operator* in a *generators* or *partitions* there must be a unique completion that makes it syntactically legal.
- There must be a unique set of completions for the abbreviated *terms* in a *trait* such that the resulting *trait* is syntactically legal.

Normalization

- Replace each abbreviated *operator* and *term* by its unique legal completion.

2.8. Mixfix Operators and Bracketing

Replace the production for *term* by:

term	::=	*logicalTerm* | **if** *term* **then** *term* **else** *term*
logicalTerm	::=	*equalityTerm* { *logicalOp equalityTerm* }*
equalityTerm	::=	*simpleOpTerm* [*eqOp simpleOpTerm*]
simpleOpTerm	::=	*simpleOp*⁺ *secondary* | *secondary simpleOp*⁺
		| *secondary* { *simpleOp secondary* }*
secondary	::=	*primary* | [*primary*] *bracketed* [**:** *sort*] [*primary*]
bracketed	::=	*openSym* [*term* { { *sepSym* | **,** } *term* }*] *closeSym*
primary	::=	{ **(** *term* **)** | *simpleId* [**(** *term*⁺**,** **)**] } { **.** *simpleId* | **:** *sort* }*

Context-sensitive checking

- In any *logicalTerm* or *simpleOpTerm* of the form $t_0\ op_1 \ldots op_n\ t_n$, the op_i must all be the same *logicalOp* or *simpleOp*.

Normalization

Mixfix *terms* are translated by creating a function application for each mixfix operator occurrence. The translated *name* of the *operator* is an *opForm* derived by replacing each subterm by "__". Unless the *operator* is a constant, this is followed by a parenthesized list of translated subterms. Grouping parentheses—those in a *primary* of the form (*term*)—are discarded. For example, the mixfix *term*

$$((\textbf{if}\ p \land q \land r\ \textbf{then}\ s \cup \{e\}\ \textbf{else}\ S[i]) \cap T)$$

is translated to the functional *term*

$$\_\_\cap\_\_(\textbf{if\_\_then\_\_else\_\_}(\_\_\land\_\_(\_\_\land\_\_(p,\ q),\ r),\ \_\_\cup\_\_(s,\ \{\_\_\}(e)),\ \_\_[\_\_](S,\ i)),\ T)$$

2.9. Implicit Markers

Definition

- A *name* is *markable* if it is a *simpleOp* or **.** *simpleId* and it appears
  - in an *operator*, or
  - as a *newOp* that renames an *operator* whose *name* contains a single *simpleOp* or **.** *simpleId*.

Context-sensitive checking

- There must be a unique marking of each markable *name* by adding one or two "__"s, such that the resulting *trait* is syntactically legal, and
  - if the *name* appears in a *renaming* in a *traitRef*, the normalization of the resulting *traitRef* is syntactically legal.
  - if the *name* appears as a *newOp* in a *renaming*, the *newOp*'s and *oldOp*'s markings have "__"s in the same positions.

Normalization

- Replace each markable *name* by its unique legal marking.

2.10. Built-in Operators

- Each explicit *trait* implicitly includes a *trait* with the *traitBody*

  introduces

  `true`: $\rightarrow$ `Bool`

  `false`: $\rightarrow$ `Bool`

  $\neg$__: `Bool` $\rightarrow$ `Bool`

  __$\wedge$__: `Bool`, `Bool` $\rightarrow$ `Bool`

  __$\vee$__: `Bool`, `Bool` $\rightarrow$ `Bool`

  __$\Rightarrow$__: `Bool`, `Bool` $\rightarrow$ `Bool`

  asserts

  `Bool` **generated by** `true`, `false`

  $\forall$ *b:* `Bool`

  $\neg$`true` == `false`

  $\neg$`false` == `true`

  `true` $\wedge$ b == b

  `false` $\wedge$ b == `false`

  `true` $\vee$ b == `true`

  `false` $\vee$ b == b

  `true` $\Rightarrow$ b == b

  `false` $\Rightarrow$ b == `true`

- For each sort `S` in an explicit *trait*'s sort set, it implicitly includes a *trait* with the *traitBody*

  introduces

  __=__: `S`, `S`$\rightarrow$ `Bool`

  __$\neq$__: `S`, `S`$\rightarrow$ `Bool`

  if__then__else__: `Bool`, `S`, `S` $\rightarrow$ `S`

  asserts

  `S` **partitioned by** =

  $\forall$ *x, y, z:* `S`

  $x = x$ == `true`

  $x = y$ == $y = x$

  $(x = y \wedge y = z) \Rightarrow x = z$ == `true`

  $x \neq y$ == $\neg(x = y)$

  if `true` **then** x **else** y == x

  if `false` **then** x **else** y == y

2.11. Boolean Terms as Equations

Add to the grammar the production:

equation	::=	*term*

Normalization

- Replace each *equation* of the form *term* by *term* == `true`.

2.12. Shorthands

Add to the grammar the productions:

traitContext	::=	*shorthand*
shorthand	::=	*enumeration* | *tuple* | *union*
enumeration	::=	*sort* **enumeration of** *elementId*$^{+}$**,**
elementId	::=	*simpleId*
tuple	::=	*sort* **tuple of** *fields*$^{+}$**,**
union	::=	*sort* **union of** *fields*$^{+}$**,**
fields	::=	*fieldId*$^{+}$**, :** *sort*
fieldId	::=	*simpleId*

Context-sensitive checking

- No *elementId* may occur more than once in an *enumeration*.
- No *fieldId* may occur more than once in a *tuple* or *union*.
- No *sort* in a *fields* may be the *sort* of the enclosing *tuple* or *union*.

Normalization

- Replace each *fields* of the form $f_1, \ldots, f_n$: S by f_1: S, $\ldots$, f_n: S.
- For each *enumeration* of the form S **enumeration of** $c_1, \ldots, c_n$
  Include in the enclosing *trait* a *trait* with the *traitBody*
  introduces
  $c_1, \ldots, c_n$: $\to$ S
  succ: S $\to$ S **asserts**
  S **generated by** $c_1, \ldots, c_n$
  equations
  $c_i \neq c_j$
  $succ(c_i) == c_{i+1}$
  for $1 \leq i < j \leq$ n

- For each *tuple* of the form S **tuple of** $\mathtt{f}_1$: $\mathtt{S}_1$, ..., $\mathtt{f}_n$: $\mathtt{S}_n$
  Include in the enclosing *trait* a *trait* with the *traitBody*

  introduces

  [__, ..., __]: $\mathtt{S}_1$, ..., $\mathtt{S}_n$ $\rightarrow$ S

  .$\mathtt{f}_i$: S $\rightarrow$ $\mathtt{S}_i$

  set_$\mathtt{f}_i$: S, $\mathtt{S}_i$ $\rightarrow$ S

  asserts

  S **generated by** [__, ..., __]

  S **partitioned by** .$\mathtt{f}_1$, ..., .$\mathtt{f}_n$

  $\forall$ s : S, x_1, y_1: $\mathtt{S}_1$, ..., x_n, y_n: $\mathtt{S}_n$

  $[x_1, \ldots, x_i, \ldots, x_n]$.$\mathtt{f}_i$ == x_i

  set_$\mathtt{f}_i$($[x_1, \ldots, x_i, \ldots, x_n]$, y_i) == $[x_1, \ldots, y_i, \ldots, x_n]$

  for $1 \leq i \leq$ n .

- For each *union* of the form S **union of** $\mathtt{f}_1$: $\mathtt{S}_1$, ..., $\mathtt{f}_n$: $\mathtt{S}_n$
  Include in the enclosing *trait* a *trait* with the *traitBody*

  S_tag **enumeration of** $\mathtt{f}_1$, ..., $\mathtt{f}_n$

  introduces

  $\mathtt{f}_i$: $\mathtt{S}_i$ $\rightarrow$ S

  .$\mathtt{f}_i$: S $\rightarrow$ $\mathtt{S}_i$

  tag: S $\rightarrow$ S_tag

  asserts

  S **generated by** $\mathtt{f}_1$, ..., $\mathtt{f}_n$

  S **partitioned by** .$\mathtt{f}_1$, ..., .$\mathtt{f}_n$, tag

  $\forall$ x_1: $\mathtt{S}_1$, ..., x_n: $\mathtt{S}_n$

  $\mathtt{f}_i(x_i)$.$\mathtt{f}_i$ == x_i

  tag($\mathtt{f}_i(x_i)$) == $\mathtt{f}_i$

  for $1 \leq i \leq$ n.

- Finally, remove each *shorthand*.

Appendix I: Logical Details

A *theory* is a set of closed formulas (formulas without free variables) in typed first-order logic with equality. Each theory contains the conventional axioms of typed first-order logic with equality, and is closed under derivability with the conventional first-order rules of inference, and thus is closed under the usual notion of semantic consequence.

Theories are formulated using a universal alphabet, rather than the smaller alphabets occurring in individual *traits*, so that the schema associated with a *generators* does not depend on the enclosing *trait*.

The *universal closure* of a formula P is $\forall x_1, \ldots, \forall x_n P$, where $x_1, \ \ldots, x_n$ are all the free variables in P.

The *substitution* of a formula e for a variable x in a formula P, denoted by $P[x \leftarrow e]$, is the result of simultaneously replacing every free occurrence of x in P by e, after renaming the bound variables as needed to avoid the capture of free variables in e.

An example of the induction schema for "`Set` **generated by** { }, `insert`" for a binary predicate P, whose first argument is of sort `Set`, is the formula

$$\forall y \Big((\forall x P(x,y)) \ \equiv \ \big(P(\{\}, y) \ \wedge \ \forall z \forall i (P(z,y) \ \Rightarrow \ P(\texttt{insert}(i,z), y))\big)\Big)$$

The formula for "`Set` **partitioned by** $\in$" is

$$\forall x \forall y \big(x = y \ \equiv \ \forall i (i \in x = i \in y)\big)$$

Appendix II: Lexical Structure

LSL was designed for use with an open-ended collection of programming languages, support tools, and input/output facilities, each of which may have its own lexical conventions and capabilities. To avoid conflicts, LSL assigns fixed meanings to only a small number of characters. To conform to local conventions and to exploit locally available capabilities, LSL's character and token classes are open-ended, and can be tailored by *initialization files*.

There are several semantically equivalent forms of LSL. Any of these forms can be mechanically translated into any other without losing information. *Interchange form* is an encoding of LSL using a subset of the ASCII character set. Characters outside this subset are represented by extended characters. Interchange form is the "lowest common denominator" for LSL. *Presentation forms* are used in environments with rich sets of characters, including this report. *Interactive forms* are used by Larch editors, browsers, checkers, etc., for input and output.

Contiguous sequences of identifier characters and contiguous sequences of operator characters form single tokens. Whitespace characters are insignificant except for separating tokens. Each of the remaining characters constitutes a separate token.

Character classification: Each character (or extended character) is classified as one of *idChar*, *opChar*, *whiteChar*, *extensionChar*, or *singleChar*. *whiteChar* contains blank, tab, and end-of-line. The required members of the other character classes are

idChar	ABCDEFGHIJKLMNOPQRSTUVWXYZ
idChar	abcdefghijklmnopqrstuvwxyz
idChar	0123456789
idChar	_
opChar	* + − . / < = >
extensionChar	\
singleChar	, : ( )

Unassigned characters can be assigned to any character class by a line in the initialization file like those above: the name of a class followed by characters to be assigned to it (possibly separated by *whiteChar*s). Assigned characters cannot be reassigned. Characters that have not been explicitly assigned are classified as *singleChar*s.

Extended characters start with an *extensionChar*. If the character following the *extensionChar* is an *idChar*, a comma, a colon, or a parenthesis, the extended character includes all following contiguous *idChar*s; otherwise it extends only through the next character (which must be a visible character). The entire extended character is classified as though it were a character; if it has not been assigned, it is classified as a *singleChar*.

Unlike other character classes, assignment of a new *extensionChar* returns the previous *extensionChar* to unassigned status. Extended characters—even those classified as *idChars*—are not included in other extended characters.

The special class *endCommentChar* initially contains end-of-line. Any real character may be assigned to this class, but extended characters cannot. It is the only character class that is not disjoint from each of the others.

Token formation: Contiguous sequences of *idChars* and contiguous sequences of *opChars* form single tokens. *whiteChars* are insignificant except for separating tokens. Each *singleChar* constitutes a separate token.

Token translation: A token may be defined as a *synonym* for another token by including a line in the initialization file of the form

synonym *oldToken newToken*

All occurrences of *newToken* are translated to *oldToken*.

Token classification: The initial members of the token classes are

quantifierSym	\forall
logicalOp	\and \or \implies
eqOp	\eq \neq
equationSym	\equals
eqSepSym	\eqsep
selectSym	\select
openSym	\(
sepSym	\,
closeSym	\)
simpleId	\:
mapSym	\arrow
markerSym	\marker
commentSym	\comment

Unassigned tokens can be assigned to any token class by a line in the initialization file like those above: the name of a class followed by tokens to be assigned to it. Assigned tokens cannot be reassigned. Any tokens in a trait that have not been explicitly assigned are classified according to the following rules:

- If the token is a sequence of *idChars* that occurs as a terminal symbol of the grammar (a *keyword*), then that symbol.
- If the token is any other sequence of *idChars*, then *simpleId*.

- If the token is a *singleChar* that occurs as a terminal symbol of the grammar (comma, colon, or parenthesis), then that symbol.
- If the token is a sequence of *opChar*s, then *simpleOp*.
- If the token is an extended character starting with an opening parenthesis, such as "\(large", then *openSym*.
- If the token is an extended character starting with a comma, then *sepSym*.
- If the token is an extended character starting with a closing parenthesis, then *closeSym*.
- If the token is an extended character starting with a colon, then *simpleId*.
- Otherwise, *simpleOp*.

If the token is classified as a *commentSym*, then it and all following characters up through the first occurrence of an *endCommentChar* are discarded, like *whiteChar*s.

Initialization: The initialization file is processed before any traits. The extensions on each line are effective on all subsequent lines.

Sample initialization files: The following initialization file would be suitable for the presentation form used this report.

<table>
<tr><td>idChar</td><td>'</td><td></td></tr>
<tr><td>opChar</td><td>¬ ! # $ & ? @ | ∈</td><td></td></tr>
<tr><td>openSym</td><td>[{ ⟨</td><td></td></tr>
<tr><td>sepSym</td><td>;</td><td></td></tr>
<tr><td>closeSym</td><td>] } ⟩</td><td></td></tr>
<tr><td>selectSym</td><td>.</td><td></td></tr>
<tr><td>synonym</td><td>\and</td><td>∧</td></tr>
<tr><td>synonym</td><td>\or</td><td>∨</td></tr>
<tr><td>synonym</td><td>\implies</td><td>⇒</td></tr>
<tr><td>synonym</td><td>\not</td><td>¬</td></tr>
<tr><td>synonym</td><td>\eq</td><td>=</td></tr>
<tr><td>synonym</td><td>\neq</td><td>≠</td></tr>
<tr><td>synonym</td><td>\arrow</td><td>→</td></tr>
<tr><td>synonym</td><td>\marker</td><td>__</td></tr>
<tr><td>synonym</td><td>\equals</td><td>==</td></tr>
<tr><td>synonym</td><td>\comment</td><td>%</td></tr>
</table>

The following initialization file would be suitable for a form limited to the ASCII character set, allowing upper-case reserved words.

```
idChar      '
opChar      ~ ! # $ & ? @ |
openSym     [ { \<
sepSym      ;
closeSym    ] } \>
selectSym   .
synonym     \and          &
synonym     \or           |
synonym     \implies      =>
synonym     \not          ~
synonym     \eq           =
synonym     \neq          ~=
synonym     \arrow        ->
synonym     \marker       __
synonym     \equals       ==
synonym     \comment      %
synonym     asserts       ASSERTS
synonym     assumes       ASSUMES
synonym     by            BY
synonym     converts      CONVERTS
synonym     else          ELSE
synonym     enumeration   ENUMERATION
synonym     equations     EQUATIONS
synonym     exempting     EXEMPTING
synonym     for           FOR
synonym     generated     GENERATED
synonym     if            IF
synonym     includes      INCLUDES
synonym     introduces    INTRODUCES
synonym     implies       IMPLIES
synonym     of            OF
synonym     partitioned   PARTITIONED
synonym     then          THEN
synonym     trait         TRAIT
synonym     tuple         TUPLE
synonym     union         UNION
```

Appendix III: Grammatical Notation

|	alternative separator
{ e }	e as a syntactic unit
[e]	optional e
e*	zero or more e's
e***,**	zero or more e's, separated by commas
e^+	one or more e's
e^+**,**	one or more e's, separated by commas
alpha	the nonterminal symbol alpha
alpha	the reserved word alpha
, : ( )	the reserved comma, colon, and parenthesis characters

For readability of grammars throughout this report, certain tokens are used to denote symbol classes—although these particular tokens are *not* reserved, and could be assigned differently by an initialization file. The correspondence is as follows:

.	*selectSym*
→	*mapSym*
__	*markerSym*
==	*equationSym*
∀	*quantifierSym*

Acknowledgments

The Larch Shared Language has evolved over many years. We have freely borrowed the best ideas we could find in other specification languages, and have received helpful criticism and suggestions from too many people to enumerate here. We are especially grateful for long-term encouragement and advice from our colleagues at MIT and DEC/SRC and from members of IFIP Working Group 2.3 (Programming Methodology). Martin Abadi, Steve Garland, Bill McKeeman, Jim Saxe, and Jeannette Wing have made important contributions to the recent evolution of the language.

Part of this work was supported at MIT by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, and by the National Science Foundation under grant CCR-8910848.

References

[Bidoit 1988] M. Bidoit, *The stratified loose approach: A generalization of initial and loose semantics,* Rapport de Recherche no. 402, Orsay, France, 1988.

[Burstall and Goguen 1980] R.M. Burstall and J.A. Goguen, "Semantics of CLEAR, a Specification Language," *Proc. Advanced Course on Abstract Software Specifications,* D. Bjorner (ed.), Springer-Verlag Lecture Notes in Computer Science 86, pp. 292–332, 1980.

[Dahl, Langmyhr, and Owe 1986] O.-J. Dahl, D.F. Langmyhr, and O. Owe, *Preliminary Report on the Specification and Programming Language ABEL,* Research Report 106, Institute of Informatics, University of Oslo, Norway, 1986.

[Ehrig and Mahr, 1985] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics,* EATCS Monographs on Theoretical Computer Science, vol. 6, Springer-Verlag, 1985.

[Garland, Guttag, and Horning 1990] S.J. Garland, J.V. Guttag, and J.J. Horning, "Debugging Larch Shared Language Specifications," *IEEE Trans. Software Engineering,* to appear; also available as Digital Equipment Corporation Systems Research Center Report 60, 1990.

[Gaudel 1985] M.-C. Gaudel, "Towards Structured Algebraic Specifications," Esprit Technical Week, Brussels, *Esprit 85 Status Report,* pp. 493–510, North-Holland, 1985.

[Goguen, Thatcher, and Wagner 1978] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," *Current Trends in Programming Methodology IV: Data Structuring,* R. Yeh (ed.), pp. 80–144, Prentice-Hall, 1978.

[Guttag 1975] J.V. Guttag, *The Specification and Application to Programming of Abstract Data Types,* Ph.D. dissertation, Computer Science Department, University of Toronto, Canada, 1975.

[Guttag and Horning 1978] J.V. Guttag and J.J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica,* vol. 10, pp. 27–52, 1978.

[Guttag and Horning 1986] J.V. Guttag and J.J. Horning, "Report on the Larch Shared Language," *Science of Computer Programming,* vol. 6, pp. 103–134, 1986.

[Guttag, Horning, and Wing 1985a] J.V. Guttag, J.J. Horning, and J.M. Wing, "The Larch Family of Specification Languages," *IEEE Software,* vol. 2, no. 5, pp. 24–36, 1985.

[Guttag, Horning, and Wing 1985b] J.V. Guttag, J.J. Horning, and J.M. Wing, *Larch in Five Easy Pieces,* Digital Equipment Corporation Systems Research Center Report 5, 1985.

[Liskov and Guttag 1986] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development,* MIT Press and McGraw-Hill Book Company, 1986.

[Sanella and Tarlecki 1987] D.T. Sanella and A. Tarlecki, "On Observational Equivalence and Algebraic Specifications," *J. Computer and System Sciences,* vol. 34, pp. 150–178, 1987.

[Wand 1979] M. Wand, "Final Algebra Semantics and Data Type Extensions," *Journal of Computer and System Sciences,* vol. 19, no. 1, pp. 27–44, 1979.

[Wirsing 1989] M. Wirsing, *Algebraic Specification,* Technical Report MIP - 8914, University of Passau, Germany, 1989.

LSL 2.3 Reference Grammar

```
trait          ::=  simpleId [ ( { name [ : signature ] }+, ) ] : trait
                    { shorthand | external }* opPart* propPart* [ consequences ]
name           ::=  simpleId | opForm
opForm         ::=  if __ then __ else __
                    | [ __ ] { simpleOp | logicalOp | eqOp }[ __ ]
                    | [ __ ] openSym [ placeList ] closeSym [ __ ]
                    | [ __ ] . simpleId
placeList      ::=  __ { { sepSym | , } __ }*
signature      ::=  sort*, → sort
sort           ::=  simpleId
shorthand      ::=  enumeration | tuple | union
enumeration    ::=  sort enumeration of simpleId+,
tuple          ::=  sort tuple of fields+,
union          ::=  sort union of fields+,
fields         ::=  simpleId+, : sort
external       ::=  { includes | assumes } traitRef+,
traitRef       ::=  { simpleId | ( simpleId+, ) } [ ( renaming ) ]
renaming       ::=  replace+, | name+, { , replace }*
replace        ::=  name for name [ : signature ]
opPart         ::=  introduces opDcl+
opDcl          ::=  name+, : signature
propPart       ::=  asserts genPartition* eqPart
genPartition   ::=  sort { generated | partitioned } by operator+,
operator       ::=  name [ : signature ]
eqPart         ::=  [ equations eqSeq ] { ∀ varDcl+, eqSeq }*
varDcl         ::=  simpleId+, : sort
eqSeq          ::=  equation { eqSepSym equation }*
equation       ::=  term [ == term ]
term           ::=  logicalTerm | if term then term else term
logicalTerm    ::=  equalityTerm { logicalOp equalityTerm }*
equalityTerm   ::=  simpleOpTerm [ eqOp simpleOpTerm ]
simpleOpTerm   ::=  simpleOp+ secondary
                    | secondary simpleOp+
                    | secondary { simpleOp secondary }*
secondary      ::=  primary | [ primary ] bracketed [ : sort ] [ primary ]
bracketed      ::=  openSym [ term { { sepSym | , } term }* ] closeSym
primary        ::=  { ( term ) | simpleId [ ( term+, ) ] } { . simpleId | : sort }*
consequences   ::=  implies { traitRef*, genPartition* eqPart
                              | [ traitRef+, genPartition+ ] eqSeq } conversion*
conversion     ::=  converts operator+, [ exempting [ ∀ varDcl+, ] term+, ]
```