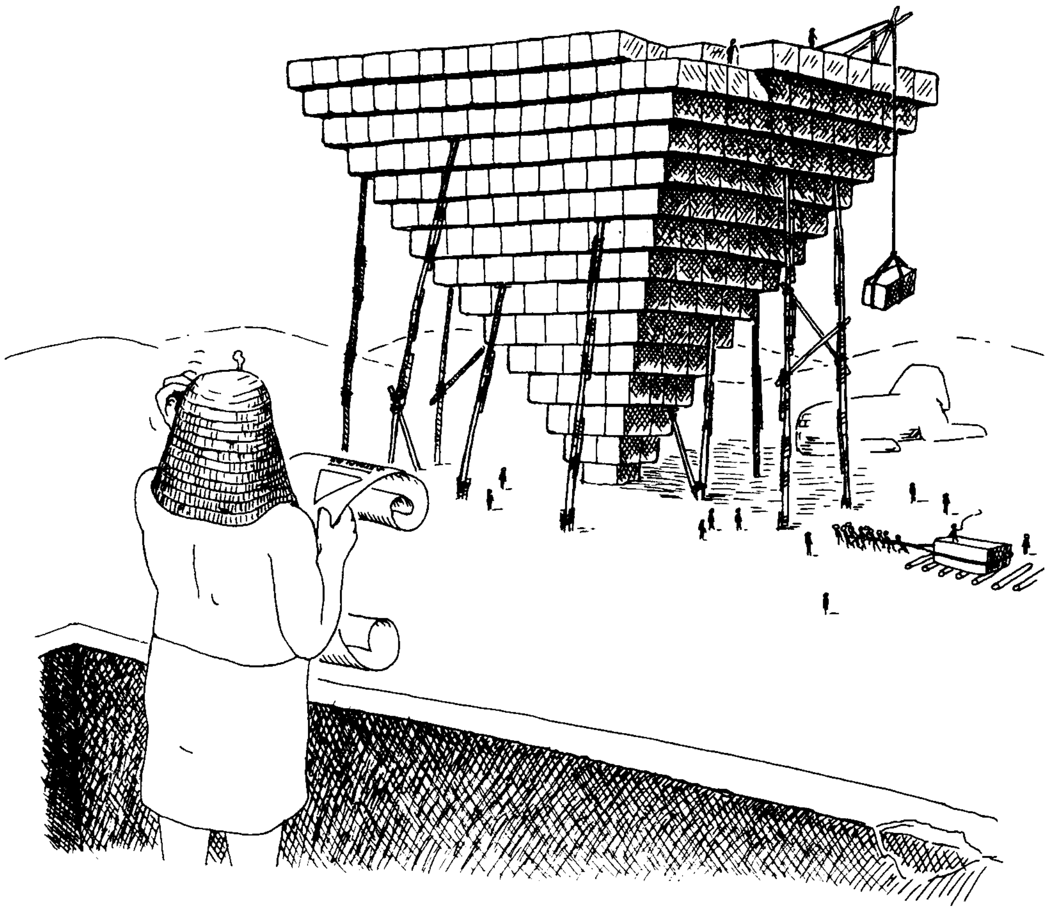
Larch in Five Easy Pieces

J. V. Guttag, J. J. Horning, and J. M. Wing



The Larch Project and related research have been supported at MIT's Laboratory for Computer Science by DARPA under contract N00014-75-C-0661 and by the National Science Foundation under Grant MCS-811984 6, at the Digital Equipment Corporation Systems Research Center and at the Xerox Palo Alto Research Center by corporate funds, and at USC by the National Science Foundation under Grant ECS-8403905.

Authors' abstract:

The Larch Project is developing tools and techniques intended to aid in the productive use of formal specifications. A major part of the Larch Project is a family of specification languages. Each Larch specification has one component written in a language derived from a programming language and another component written in a language independent of any programming language. We call the former *Larch interface languages* and the latter the *Larch Shared Language.* We have gathered together five documents about the Larch family of languages: an overview, an informal description of the Shared Language, a reference manual for the Shared Language, a handbook of specifications written in the Shared Language, and a report on using Larch/CLU, which is one of the interface languages.

J. V. Guttag, J. J. Horning, and J. M. Wing

Capsule review:

The Larch approach is geared towards specifying program modules (as if defining abstract data types) to be implemented in particular programming languages. Predicate-oriented *interface languages* are used to describe the intended behaviour of procedures. Abstractions are formulated in the *Shared Language.* Descriptions given in the interface languages are given in terms of those abstractions and might also include descriptions of error-reactions and implementation limits.

Similar in appearance to many algebraic specification languages, the Shared Language can be used for specifying abstract data types, but its focus is on specifying "smaller" entities or properties (such as commutativity, group theory, and generic properties of container-like types). Such entities are expressed as independent, tractable, and reusable building blocks.

The Shared Language offers a simple, syntactic approach to modularization and composition. Units of specifications, called traits, are combined by syntactic inclusion; inclusions can be equipped with renaming rules. Traits are never explicitly parameterized; the renaming mechanism makes any entity of a trait a potential parameter. The meaning of a trait is a first-order theory. It is obtained as the conservative union of the theories associated with included traits and the set of local axioms of a trait. The local axioms are expressed as first-order, quantified equations. The language allows – and the design philosophy encourages – redundant theorems to be stated, thus enabling considerable amounts of consistency checking to be done (possibly by mechanical theorem proving).

The report contains introductory, motivating, and reference information. A number of sample shared language specifications and small examples of CLU and Pascal interface language specifications are given. Also included is a major (CLU) example describing, step by step, how pieces of shared and interface language specification are constructed. The reference material consists of a terse reference manual, which defines the shared language as a kernel and its syntactic extensions, and a 'handbook' of often-used abstractions, such as group theory, lattices, sets, stacks, queues, mappings, and graphs.

Søren Prehn

Contents:

Prelude

The Larch Project

The Larch Project at MIT's Laboratory for Computer Science and DEC's Systems Research Center is the continuation of collaborative research into the uses of formal specifications that started with the work reported in [Guttag 75]. The project is developing both a family of specification languages and a set of tools to support their use, including language-sensitive editors and semantic checkers based on a powerful theorem prover.

Larch is an effort to test our ideas about making formal specifications useful. To focus the project, we made some assumptions that strongly influenced the directions it has taken:

- *Local specifications.* We started with the belief that behavioral specifications of components of sequential programs could be useful in the near future. No conceptual breakthroughs or theoretical advances seemed to be needed. Rather, we needed to use what we already knew to design usable languages, develop some software support tools, and educate some system designers and implementers.
- *Errors.* Our experience suggests that the process of writing specifications can be as error-prone as the process of programming. We believe, therefore, that it is important to do a substantial amount of checking of the specifications themselves. There are two ways to detect errors: human inspection and mechanical checking. We want our specification languages to make it easy to write readable specifications. We also want them to incorporate redundancy that will allow mechanical checks to detect many common errors.
- *Scale.* We want methods that are useful even when there are many requirements to be recorded in a specification. Methods that are entirely adequate for one-page specifications may fail utterly for hundred-page specifications. It is essential that large specifications be composed from small ones that can be understood separately, and that the task of understanding the ramifications of their combination be manageable. For large specifications, as pointed out by [Burstall and Goguen 77], the "putting together" operations are more crucial than the details of the language used for the pieces.
- *Incremental construction.* Large specifications, like large programs, must be constructed incrementally. Most specifications are unfinished during most of their useful lifetime. Consequently, it is essential to reason about and to check unfinished specifications.
- *Incompleteness.* Many finished specifications are incomplete. Sometimes this incompleteness is caused by abstraction from details that are irrelevant for a particular

purpose; for example, time, storage usage, and functionality might be specified separately. Sometimes, however, it is a symptom of oversights in the design or specification process. A specification checker should be able to distinguish between oversights and intentional incompleteness.

- *Tools.* We believe that tools have an important role to play in the specification process. The number of tedious and error-prone tasks associated with maintaining a substantial body of formal text in a consistent state is a serious bar to the practical use of formal specifications. Tools can assist in managing the sheer bulk of large specifications, in browsing through selected pieces, in detecting errors, in deriving interactions and consequences, and in teaching a new methodology. Languages designed to exploit powerful tools may be quite different from pencil-and-paper languages.

- *Reusability.* It is inefficient to start each specification from scratch. We do not want to keep reinventing the specifications of integers and sets—or even priority queues and bitmaps. We need a repository of reusable specification components that have evolved to handle the common cases well, and that can serve as models when we are faced with uncommon cases. The collection should be open-ended, and include application-oriented abstractions, as well as mathematical and implementation-oriented ones.

- *Language dependencies.* The environment in which a program component is embedded, and hence the nature of its observable behavior, is likely to depend in fundamental ways on the semantic primitives of the programming language. Any attempt to disguise this dependence will make specifications more obscure to both the component's clients and its implementers. On the other hand, many of the important abstractions in most specifications can be defined independently of any programming language.

Piece I

The Larch Family of Specification Languages

1. Introduction

For well over a decade, researchers have suggested that the use of formal specification techniques could play a valuable role in the development of software. Although there has been considerable progress in developing a theoretical basis for such specifications, practical experience is rather limited. This report describes the current state of a research project intended to have practical applications in the next few years.

The Larch Project is developing tools and techniques to aid in the productive application of formal specifications. It is based upon a *two-tiered* approach to specification. Each Larch specification has components written in two languages: one designed for a specific programming language and another common to all programming languages. We call the former *Larch interface languages*, and the latter the *Larch Shared Language.*

We use interface languages to specify program components. Each interface specification should provide the information needed to write programs that use the specified component. A critical part of this interface is how the component communicates with its environment. Communication mechanisms differ from programming language to programming language, sometimes in subtle ways. We have found that it is easier to be precise about communication when the specification language reflects the programming language. Such specifications are generally shorter than those written in a "universal" interface language. They also seem to be 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 provides a way to write assertions about program states; these assertions can be translated to predicate calculus formulas. It incorporates programming-language-specific notations for constructs such as side effects, exception handling, and iterators. Its simplicity or complexity depends largely upon the simplicity or complexity of the observable state and state transformations of its programming language.

Larch is intended to support a style of program design in which data abstractions play a prominent role. Each interface language has a mechanism for specifying data abstractions. If its programming language provides direct support for data abstractions, the interface language facility is modeled on that of the programming language; if it does not, the facility is designed to be compatible with other aspects of the programming language.

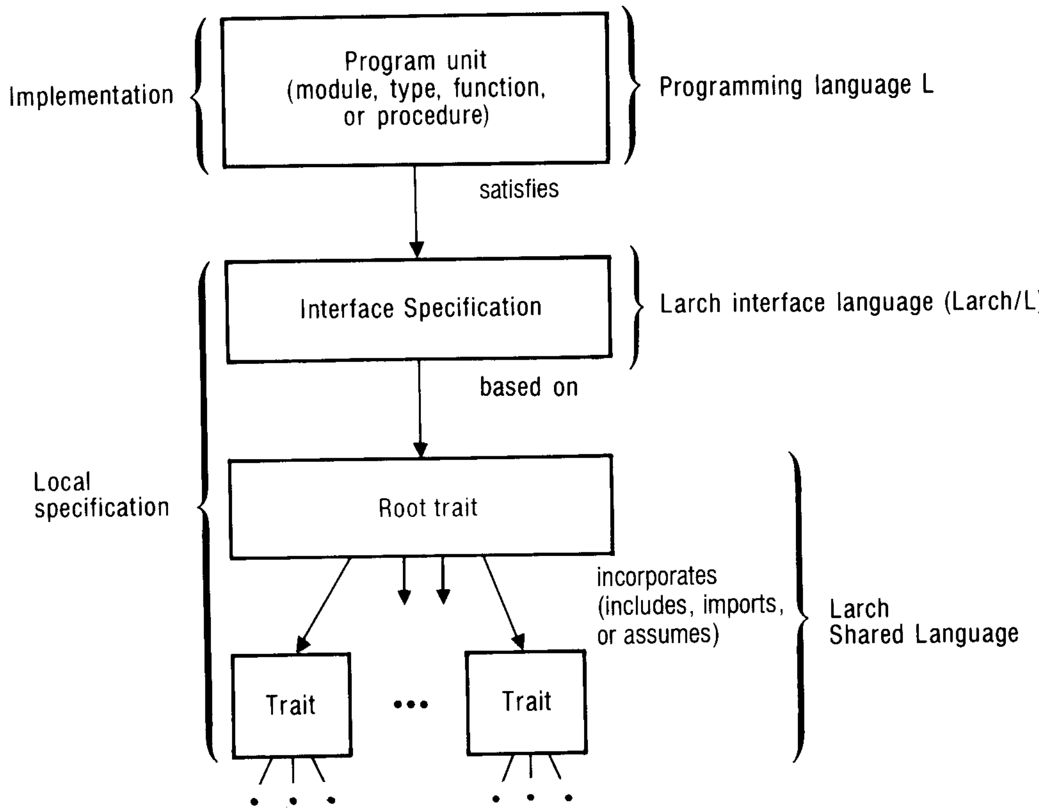


Figure 1. Two-Tiered Specification in Larch

The Shared Language is used to define terms used in interface specifications. It generates theories that are independent of any programming language. The Shared Language is primarily algebraic: equations define relations among operators, giving meaning to the notion of equality between terms that appear in interface specifications.

The two-tiered structure of Larch specifications is illustrated in Figure 1, and discussed more fully in Piece V.

Some important aspects of the Larch family of specification languages are:

- *Composability.* The Larch languages are designed for the incremental construction of specifications from other specifications.
- *Emphasis on presentation.* The Larch languages are designed to be readable. Among other things, Larch's composition mechanisms are defined as operations on specifications, rather than on theories or models [Sannella and Tarlecki 85].
- *Suitability for integrated, interactive tools.* The Larch languages are designed to facilitate the interactive construction and incremental checking of specifications.
- *Semantic checking.* The Larch languages are designed to enable extensive checking of specifications as they are being constructed. An important aspect of our approach is the use of a powerful theorem prover for semantic checking to supplement the syntactic checking commonly defined for specification languages.
- *Localized programming language dependencies.* Each Larch interface language encapsulates the features needed to write concise and comprehensible specifications for a particular programming language, and incorporates Shared Language specifications in a uniform way.

The next two sections present the Larch Shared Language and two Larch interface languages by means of a series of example specifications that also illustrate the way we expect specifications to be structured.

Section 2 contains Larch Shared Language specifications for a number of abstractions that would be useful in any programming language, culminating in specifications of the data structures `PriorityQueue` and `MultiSet`. Section 3 contains Larch/Pascal and Larch/CLU specifications of closely-related data types for Pascal and CLU. Issues such as boundedness, preconditions, and exception-handling are dealt with in ways that are appropriate to the respective programming languages. Section 4 contains some general remarks about our two-tiered approach to writing specifications.

2. The Larch Shared Language

The complete syntax and semantics of the Larch Shared Language are given in Pieces II and III. Here we present a series of short examples that introduce most of the language, a few features at a time.

The *trait* is the basic unit of specification in the Larch Shared Language. A trait introduces operators and specifies their properties. Sometimes the collection of operators will correspond to an abstract data type. Frequently, however, it is useful to define properties that do not fully characterize a type.

Our first example is a trait specifying a class of tables that store values in indexed places. It is similar to a conventional algebraic specification in the style of [Guttag and Horning 78] or [Ehrig and Mahr 85].

```
TableSpec: trait
    introduces
        new: → Table
        add: Table, Index, Val → Table
        # ∈ #: Index, Table → Bool
        eval: Table, Index → Val
        isEmpty: Table → Bool
        size: Table → Card
    constrains new, add, ∈, eval, isEmpty, size so that
        for all [ ind, ind1: Index, val: Val, t: Table ]
            eval(add(t, ind, val), ind1) = if ind = ind1 then val else eval(t, ind1)
            ind ∈ new = false
            ind ∈ add(t, ind1, val) = (ind = ind1) | (ind ∈ t)
            size(new) = 0
            size(add(t, ind, val)) = if ind ∈ t then size(t) else size(t) + 1
            isEmpty(t) = (size(t) = 0)
```

The part of the specification following **introduces** declares a set of *operators*, each with its *signature* (the *sorts* of its domain and range). These signatures are used to sort-check *terms* in much the same way as function calls are type-checked in programming languages. We use the words "operator," "sort," and "term" in describing the Larch Shared Language to avoid confusion with the similar concepts "function," "type," and "expression" in programming languages.

The final part of the specification constrains the operators by means of equations that relate terms containing them. In general, each equation involves several operators, and an operator may appear in several equations.

The first equation resembles a recursive function definition, since the operator `eval` appears on both the left and right sides. However, it does not fully define `eval`; it states a relation that must hold among `eval`, `add`, and the built-in operator **if then else**. The second and third equations together provide enough information to define the operator $\in$ (when applied to any term built up using `new` and `add`) in terms of the built-in operators `false` and $|$, and the operator $=$ for sort `Index`.

The set of theorems that can be proved about the terms defined in a trait is called its *theory*. It is the infinite set of predicate calculus formulas that consists of the trait's equations, the inequation $\neg$(`true` $=$ `false`), and all of the theorems that can be derived from these formulas plus the axioms and rules of inference of first order predicate calculus with equality.

The theory associated with `TableSpec` contains formulas that can be proved by substituting equals for equals. However, there is no meta-rule stating that if two terms are not provably equal, then they are unequal, nor is there a meta-rule stating that if two terms are not provably unequal, then they are equal. For example, we cannot determine whether `add` is permutative. The equation

$$\mathtt{add}(\mathtt{add}(t,\ ind,\ val),\ ind_1,\ val) = \mathtt{add}(\mathtt{add}(t,\ ind_1,\ val),\ ind,\ val)$$

is not in `TableSpec`'s theory, but neither is any inequation that would distinguish between the left and right hand sides. Later, we discuss Larch Shared Language constructs that can be used to generate stronger (larger) theories containing the answers to such questions.

The next series of examples defines a number of properties that are finally combined in different ways to define two traits that correspond to familiar abstract data types. Figure 2 may be used as a road map for these examples, which are presented in a bottom-up fashion, with the exception of the handbook traits `TotalOrder`, `Cardinal`, and `Equality`, which are used in the examples but not defined until Piece IV.

The trait `Container` abstracts the common properties of data structures that contain elements, such as sets, multisets, queues, and stacks. We have found it useful both as a starting point for specifications of many kinds of containers, and as an assumption when defining generic operators.

The new construct in `Container` is the **generated by** clause. It indicates that each term that does not contain any variables of sort `C` is equal to some term in which `new` and `insert` are the only operators with range `C`. Thus, it introduces an inductive rule of inference that can be used to prove properties that are true for all terms of sort `C`.

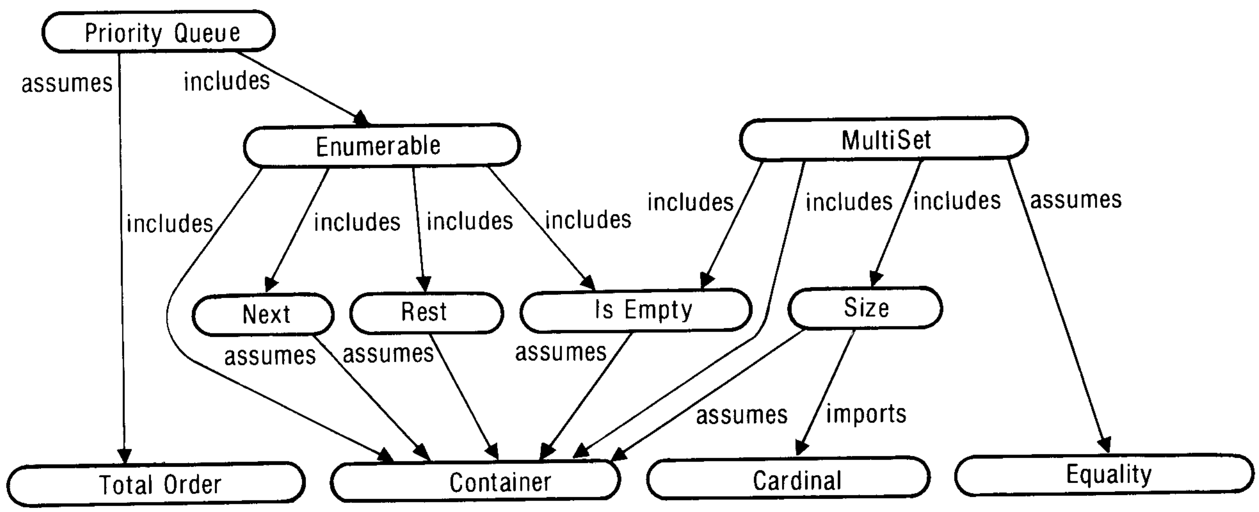


Figure 2. Relations Among the Example Traits

```
Container: trait
    introduces
        new: → C
        insert: C, E → C
    constrains C so that C generated by [ new, insert ]
```

The trait `IsEmpty` builds on `Container` by assuming it. It constrains the `new` and `insert` operators that it inherits from `Container`, as well as the operator that it introduces, `isEmpty`.

The **converts** clause in `IsEmpty` adds nothing to its theory. It adds checkable redundancy by indicating that this trait is intended to contain enough axioms to define `isEmpty`. That is, any term with no variables of sort `C` should be provably equal to one that does not contain `isEmpty`. Because of the **generated by** inherited from `Container`, this can be proved by induction, using `new` as the basis and using `insert`(c, e) in the induction step.

```
IsEmpty: trait
    assumes Container
    introduces isEmpty: C → Bool
    constrains isEmpty, new, insert so that for all [ c: C, e: E ]
        isEmpty(new) = true
        isEmpty(insert(c, e)) = false
    implies converts [ isEmpty ]
```

Next and Rest also assume Container. Like **converts**, **exempts** is present only for checking. The **exempts** clauses in Next and Rest indicate that the lack of equations for next(new) and rest(new) in these traits is intentional. Even if Next or Rest is included into a trait that claims the convertibility of next or rest, the terms next(new) and rest(new) don't have to be convertible.

```
Next: trait
    assumes Container
    introduces next: C → E
    constrains next, insert so that for all [ e: E ]
        next(insert(new, e)) = e
    exempts next(new)

Rest: trait
    assumes Container
    introduces rest: C → C
    constrains rest, insert so that for all [ e: E ]
        rest(insert(new, e)) = new
    exempts rest(new)
```

Size assumes Container, and partially defines the size operator. The phrase **imports** Cardinal means that the theory of the importing trait, Size, is a *conservative extension* of the theory of the imported trait, Cardinal. That is, Size's theory contains Cardinal's theory, but does not further constrain any of the operators appearing in Cardinal, such as 0. Consequently, the operators of Cardinal can be understood independently, since they must not be given any new properties in Size.

```
Size: trait
    assumes Container
    imports Cardinal
    introduces size: C → Card
    constrains size so that
        size(new) = 0
```

The Enumerable trait specifies properties common to containers that keep their contents in a definite order, such as stacks, queues, priority queues, sequences, and vectors. It augments Container by combining it with IsEmpty, Next, and Rest. The **includes** clause indicates that Enumerable is intended to inherit their operators and axioms and to further constrain the operators. The assumption of Container by the traits Next, Rest and IsEmpty is discharged in Enumerable by the explicit inclusion of Container.

The **partitioned by** clause indicates that next, rest, and isEmpty are sufficient to distinguish any unequal terms of sort C. Thus, for any terms t_1 and t_2, if the equalities $\texttt{next}(t_1) = \texttt{next}(t_2)$, $\texttt{rest}(t_1) = \texttt{rest}(t_2)$, and $\texttt{isEmpty}(t_1) = \texttt{isEmpty}(t_2)$ all hold, we may conclude that $t_1 = t_2$.

```
Enumerable: trait
    includes Container, Next, Rest, IsEmpty
    constrains C so that C partitioned by [ next, rest, isEmpty ]
```

PriorityQueue specializes Enumerable by further constraining next, rest, and insert. Sufficient axioms are given to convert next and rest. The axioms that convert isEmpty are inherited from the trait Enumerable, which inherited them from the trait IsEmpty.

The **with** clause in the **assumes** clause indicates that the assumed trait is TotalOrder with the sort E substituted for the sort T throughout its text.

```
PriorityQueue: trait
    assumes TotalOrder with [ E for T ]
    includes Enumerable
    constrains next, rest, insert so that for all [ q: C, e: E ]
        next(insert(q, e)) =
            if isEmpty(q) then e
            else if next(q) ≤ e then next(q) else e
        rest(insert(q, e)) =
            if isEmpty(q) then new
            else if next(q) ≤ e then insert(rest(q), e) else q
    implies converts [ next, rest, isEmpty ]
```

The final example, MultiSet, is a specialization of Container that does not satisfy Enumerable. It combines Container, IsEmpty, and Size, and introduces three new operators, count, delete, and numElements.

Constrains MSet is a shorthand for a **constrains** clause listing all the operators whose signature includes MSet. The **partitioned by** indicates that count alone is sufficient to distinguish unequal terms of sort MSet. That is, if for every term, u, count(t_1, u) = count(t_2, u), then $t_1 = t_2$.

Converts [isEmpty, count, delete, numElements, size] is a stronger assertion than the combination of an explicit **converts** [count, delete, numElements, size] with the inherited **converts** [isEmpty].

The **with** clause calls for a substitution of the operator {} for the operator new, as well as the sort MSet for the sort C.

```
MultiSet: trait
    assumes Equality with [ E for T ]
    includes IsEmpty, Size, Container with [ MSet for C, {} for new ]
    introduces
        count: MSet, E → Card
        delete: MSet, E → MSet
        numElements: MSet → Card
    constrains MSet so that
        MSet partitioned by [ count ]
        for all [ m: MSet, e1, e2: E ]
            count({}, e1) = 0
            count(insert(m, e1), e2) = count(m, e2) + (if e1 = e2 then 1 else 0)
            size(insert(m, e1)) = size(m) + 1
            numElements({}) = 0
            numElements(insert(m, e1)) =
                numElements(m) +(if count(m, e1) > 0 then 0 else 1)
            delete({}, e1) = {}
            delete(insert(m, e1), e2) =
                if e1 = e2 then m else insert(delete(m, e2), e1)
    implies converts [ isEmpty, count, delete, numElements, size ]
```

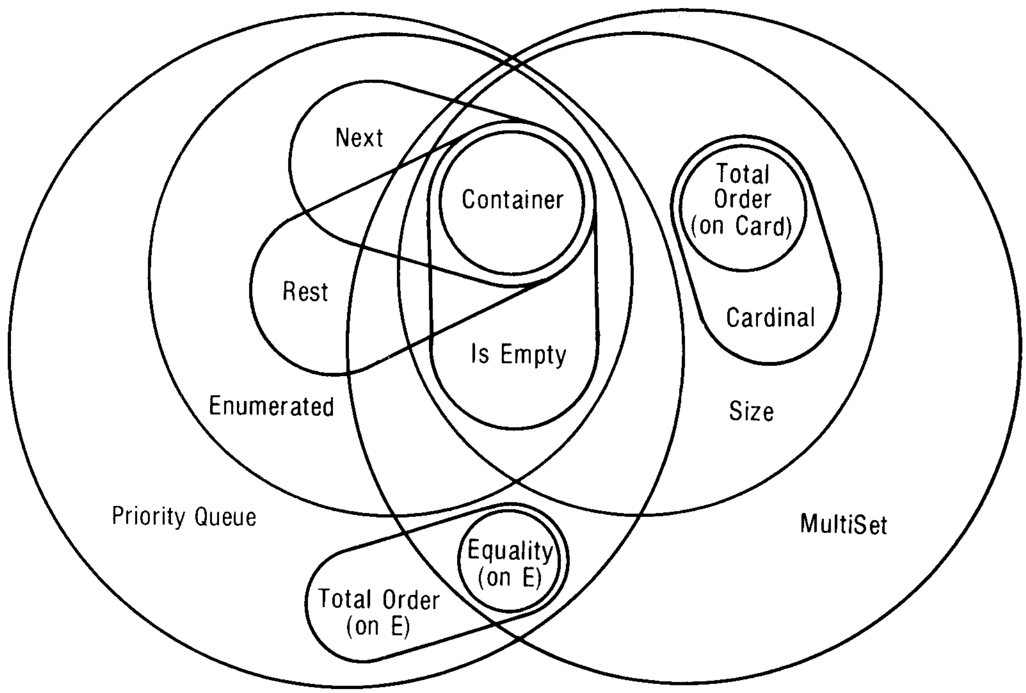


Figure 3. Inclusion Relations Among the Theories of the Example Traits

The theory associated with any trait includes the theories of each of the traits that it **assumes**, **includes**, or **imports**. Thus, Figure 3 is another way of viewing the relations among traits that were shown in Figure 2.

The theories associated with `MultiSet` and `PriorityQueue` say quite a bit about their respective data structures. These structures have much in common, and some important differences (e.g., order of insertion is significant in `PriorityQueue`, and not in `MultiSet`). Note also some things that have *not* yet been specified about these data structures. We have not specified how they are to be represented. We have not chosen the algorithms to manipulate them. We have not even said what routines are to be provided to operate on them. We have not specified how errors are to be handled. Decisions of the latter two kinds are recorded in interface specifications; the first two are in the province of the implementation.

The Shared Language examples in this report (or any other sequence of simple examples) may give a misleading image of the process of developing Larch specifications. We almost never define new abstractions starting from first principles, because traits for many of the most useful abstractions are already available. For example, the handbook in Piece IV contains the traits `Container`, `IsEmpty`, `Next`, `Rest`, `Size`, `Enumerable`, and `PriorityQueue` that have been used as examples. The handbook trait `Bag` introduces a number of operators not needed for `MultiSet`, which causes no problem. However, it is missing the operator `numElements`. In practice, we would simply **include** `Bag` in `MultiSet`, **introduce** `numElements`, and **constrain** `numElements` with two equations.

We expect Shared Language traits to be the principal reusable units in Larch. By reusing existing traits, specifiers will save time and avoid errors. Reusing traits drawn from a generally accessible handbook will also serve to standardize notation. We think of handbooks as the concentrated essence of abstractions that experienced specifiers have found useful. Piece IV contains sections on single-operator properties, binary relations, ordering relations, group theory, numeric types, simple data structures, containers, container operations, nonlinear structures, rings and fields, lattices, enumerated types, and displays. Future handbooks will specify further abstractions.

New traits are unlikely to have as much structure as is present in the various specializations of `Container` and in other parts of our handbooks. This kind of structure tends to come after a large number of related traits have been written and regularities recognized, or when the abstraction represents a well-studied mathematical system. The development of such structure represents a kind of intellectual capital that yields its dividends in future applications.

3. Larch Interface Languages

We now turn our attention to interface specifications. It is these specifications that actually describe program components that are to be implemented. The role of the Shared Language traits is to define the theories that give meaning to operators that appear in the interface specifications.

Each Larch interface language is designed for a programming language, which influences everything from the modularization mechanisms to the choice of reserved words. Larch/CLU and Larch/Pascal are presently the only two moderately well-developed Larch interface languages. A detailed description of the semantics of an early version of Larch/CLU is given in [Wing 83]. No such description of Larch/Pascal is yet available. However, a discussion of the style of Pascal programming that Larch/Pascal is designed to support is contained in [Guttag and Liskov 86].

We will illustrate each of these interface languages by means of a small example. Both Larch/Pascal and Larch/CLU support the specification of data and procedural abstractions. Since data abstractions include procedural abstractions, we organize our discussion around the former.

In both interface languages, a specification of a data abstraction (type) has three parts:

- A header giving the type name and the names of the externally visible routines,
- An associated trait and a mapping from the types in the data abstraction to sorts in the trait, and
- Interface specifications for each routine (procedure or function) of the type. A specification of a routine has three parts:
    - A header giving the name of the routine, and the names and types of its formals (parameters and returned values),
    - An associated trait providing the theory of the operators that appear in the body (in the examples, this trait is just the union of the traits associated with the types in the routine's header), and
    - A body stating any requirements on the routine's parameters and specifying the effects the routine must have when those requirements are met.

The meaning of programming language reserved words is derived directly from their meaning in the programming languages. For example, the meaning of **var** in Larch/Pascal is derived from the meaning of **var** in a Pascal parameter list; the meaning of **signals** in Larch/CLU is derived from the meaning of **signals** in CLU.

An Example Larch/Pascal Specification

Here is the Larch/Pascal specification of a data abstraction that provides a type, three procedures, and one function:

```
type Bag exports bagInit, bagAdd, bagRemove, bagChoose
    based on sort MSet from MultiSet with [ integer for E ]
    procedure bagInit(var b: Bag)
        modifies at most [ b ]
        ensures b_post = {}
    procedure bagAdd (var b: Bag; e: integer)
        requires numElements(insert(b, e)) ≤ 100
        modifies at most [ b ]
        ensures b_post = insert(b, e)

    procedure bagRemove (var b: Bag; e: integer)
        modifies at most [ b ]
        ensures b_post = delete(b, e)
    function bagChoose (b: Bag; var e: integer) : boolean
        modifies at most [ e ]
        ensures if ¬isEmpty(b)
                then bagChoose & count(b, e_post ) > 0
                else ¬bagChoose & modifies nothing
end Bag
```

The body of each routine's specification places constraints on the arguments with which the routine may properly be called, and defines the relevant aspects of the routine's behavior when it is properly called. It can be straightforwardly translated to a predicate over two states in the style of [Hehner 84] by combining its three predicates into a single predicate of the form:

Requires Predicate ⇒ (Modifies Predicate & Ensures Predicate).

An omitted **requires** is interpreted as `true`.

In the body of a Larch/Pascal specification, as in Pascal, the name of a function stands for the value returned by that function. Formal parameters may appear unqualified or qualified by *post*. An unqualified formal stands for the value of that formal when the routine is called. A formal qualified by post, such as b_{post}, stands for the value of that formal when the routine returns.

The predicate **modifies at most** [$v_1, ..., v_n$] asserts that the routine changes the value of no variable in the environment of the caller except possibly some subset of the variables denoted by the elements of $\{v_1, ..., v_n\}$. Notice that this predicate is really an assertion about all of those variables that do not appear in the list, rather than about those that do. **Modifies at most** is a built-in programming-language-specific predicate. Each Larch interface language comes equipped with its own set of built-in predicates.

The need to indicate the variables that may be modified and to distinguish between the values of variables on entry to and return from routines arises because Pascal is a language in which statements may alter memory. Since the operators in a Larch Shared Language specification represent functions, this complication does not arise there, nor would it in an interface language for a functional programming language.

In an interface specification, we give meaning to names appearing in programs by relating them to names appearing in traits. Thus it is the names in an interface specification that tie it to traits in the Larch Shared Language and to programs in its programming language. Operators (e.g., `insert`), sorts (e.g., `MSet`), and trait names (e.g., `MultiSet`) provide the link to a theory defined by a collection of traits. Names of routines (e.g., `bagAdd`), formal parameters (e.g., *e*), and types (e.g., `integer`) provide the link to programs that implement the specification. It is important not to confuse operators and sorts (from the Larch Shared Language) with routines and types (from the programming language). Operators and sorts appear in specifications, and in reasoning about specifications, but they do not appear in programs. Conversely, routines and types appear in programs, but not in traits.

The **based on** clause associates the type `Bag` with the sort `MSet` that appears in trait `MultiSet`. This association means that within this specification Shared Language terms of sort `MSet` will be used to represent Pascal values of type `Bag`. For example, the term {} is used to represent the value that *b* is to have when `bagInit` returns.

The **requires** clause of `bagAdd` states a precondition that is to be satisfied on each call. It reflects the specifier's concern with how this type can be implemented in Pascal. By putting a bound on the number of distinct elements in the `Bag`, the specification allows a fixed-size representation. It is quite natural for such considerations to surface in interface specifications; it would not be so natural for them to appear in traits.

The most interesting routine is probably `bagChoose`. Its specification is nondeterministic, because it says that bagChoose must set *e* to some value in *b* (if *b* isn't empty), but doesn't say which value. Moreover, it doesn't even require that different invocations of `bagChoose` with the same value produce the same result. The implementation we give later is abstractly nondeterministic, even though it is a deterministic program. The value to which *e* is set depends upon the order in which elements have been added to and removed from *b*, whereas this order does not affect *b*'s abstract value.

The `Bag` interface specification records a number of design decisions beyond those contained in the trait `MultiSet`. It says which routines must be implemented, and for each routine it indicates both the condition that must hold at the point of call and the condition that must hold upon return. This constitutes a contract that establishes a "logical firewall" between the implementers and the clients of type Bag. It allows them to proceed independently, relying only on the interface specification.

The clients must establish the **requires** clause at each point of call. Having done that, they may presume the truth of the **ensures** clause on return, and that only variables in the **modifies at most** clause are changed. They need not be concerned with how this happens.

The implementers are entitled to presume truth of the **requires** clause on entry. Given that, they must establish the **ensures** clause on return, while respecting the **modifies at most** clause.

Because the interface specification does not specify either the representation of the type or the algorithms in routines, yet another tier of design is needed. Because this tier is hidden from clients of the data type, the design may be changed without affecting their correctness.

The specification of each routine in an interface can be understood without reference to the specifications of other routines. This is in contrast to traits, where the specification constrains the operators by giving relations among them. Of course, to understand the type itself, to reason about it, or to design an efficient representation for it, the specifications of all its routines must be taken into account.

An Example Pascal Implementation

To illustrate the relation between an interface specification and an implementation, we give a Pascal implementation of type Bag. Neither the data structure chosen for the representation nor the program itself is very interesting. Many other implementations—some of them very different—could satisfy the same interface specification.

Both the abstraction function and the representation invariant are presented informally in this example. If we had included formal specifications of the array types used in the representation, we could have presented the abstraction function and the representation invariant formally, using a program annotation language [Luckham and von Henke 85]. Then they could be mechanically combined with the interface specifications already given to derive a concrete specification for each routine, which could then be verified separately.

Notice that the implementation of bagAdd relies on the **requires** clause of its specification.

```
const MaxBagSize = 100;

type
    ElemVals = array [1..MaxBagSize] of integer;
    ElemCounts = array [1..MaxBagSize] of integer;
    Bag = record elems: ElemVals; counts: ElemCounts; end;

{Abstraction function: the abstract bag is equivalent to the result of inserting into the
    empty bag each integer in elems a number of times equal to the corresponding
    number in counts}

{Rep invariant: each integer in counts is at least zero and no integer appears in elems
    more than once associated with a positive value in counts}

procedure bagInit(var b: Bag);
    var i: 1..MaxBagSize;
    begin
        for i := 1 to MaxBagSize do b.counts[i] := 0
    end {bagInit};

procedure bagAdd(var b: Bag; e: integer);
    var i, lastEmpty: 1..MaxBagSize;
    begin
        i := 1;
        while (i < MaxBagSize) and (b.elems[i] <> e) do
            begin
                if b.counts[i] = 0 then lastEmpty := i;
                i := i + 1
            end;
        if b.elems[i] = e then b.counts[i] := b.counts[i] + 1
        else begin
            if b.counts[i] = 0 then lastEmpty := i;
            b.elems[lastEmpty] := e;
            b.counts[lastEmpty] := 1
        end;
    end {bagAdd};
```

```
procedure bagRemove(var b: Bag; e: integer);
    var i: 1..MaxBagSize;
    begin
        i := 1;
        while (not((b.elems[i] = e) and (b.counts[i] > 0)) and (i < MaxBagSize)) do
            i := i + 1;
        if (b.elems[i] = e) and (b.counts[i] > 0) then b.counts[i] := b.counts[i] - 1
    end {bagRemove};

function bagChoose(b: Bag; var e: integer): boolean;
    var i: 1..MaxBagSize;
    begin
        i := 1;
        while (i < MaxBagSize) and (b.counts[i] = 0) do i := i + 1;
        if b.counts[i] = 0 then bagChoose := false {e not modified}
        else begin
            e := b.elems[i];
            bagChoose := true
        end
    end {bagChoose};
```

Data Types and Induction

Induction is useful in reasoning about data abstractions. There are different induction principles that can be applied on the Shared Language tier, on the interface language tier, and on the programming language tier. They are all distinct, and are useful to prove different kinds of theorems.

- Induction over a set of generating operators is used to prove theorems about *all terms* of a sort. For example, we might use it to prove by induction over `new` and `insert` that the sum of the counts of all the elements in any `MSet` is equal to its size.

- Induction over the specifications of a data type's routines (often called data type induction) is used to prove theorems about *all legal values* of a type. For example, we might show by induction over `bagInit`, `bagAdd`, and `bagRemove`, that no `Bag` has more than one hundred distinct elements. Such a proof would depend on the assumption that objects of type `Bag` are manipulated only by legal calls on the routines in `Bag`'s specification. Although this restriction is not enforced by the Pascal language, we could adopt it as a programming convention [Guttag and Liskov 85].

- Induction over the implementations of a type's routines is outside the domain of interface specifications and into that of program verification. It is used to prove theorems about *all computations* or *all reachable representations* of a type, for example, to prove that a representation invariant is established and preserved.

An Example Larch/CLU Specification

Now we use Larch/CLU to specify a bag type. The abstraction is different from the one specified in Larch/Pascal because it exploits features of CLU that do not have analogs in Pascal. But it is based on the same Larch Shared Language trait.

```
bag mutable type exports init, add, remove, choose
    based on sort MSet from MultiSet with [ int for E ]
    init = proc() returns(b: bag)
        modifies nothing
        ensures new(b) & b = {}
    add = proc(b: bag, e: int)
        modifies at most [ b ]
        ensures b_post = insert(b, e)
    remove = proc(b: bag, e: int)
        modifies at most [ b ]
        ensures b_post = delete(b, e)
    choose = proc(b: bag) returns(e: int) signals (empty)
        modifies nothing
        ensures
            normally count(b, e) > 0 except
            signals empty when isEmpty(b)
```

This example illustrates some of the ways in which programming language dependencies influence interfaces, specifications, and interface languages. Some of the programming language dependencies are trivial: the syntax has been changed to resemble that of CLU, and routine names don't start with "bag," since in CLU all calls are prefixed with the type name. Some of the dependencies, however, are more substantial.

In the body of a Larch/CLU specification, an unqualified argument formal stands for the value of the object bound to that argument on entry to the routine. An unqualified result formal stands for the value of the object bound to that argument on exit from the routine.

New is a Larch/CLU built-in predicate. The constraint new(b) means that the object bound to b when the routine returns must be distinct from all previously accessible objects. Thus, init must not return an alias for an existing bag. Larch/Pascal has a built-in predicate with a similar meaning, but it is used less often because fewer Pascal interfaces deal with dynamically allocated variables.

The built-in types of CLU, unlike those of Pascal, offer no incentive to place an *a priori* bound on the size of objects. Thus there is no **requires** clause in the specification of add.

The use of **signals** is another CLU-specific aspect of the specification. The CLU choose has a rather different header than does the Pascal bagChoose. CLU interfaces are typically designed to use CLU's exception handling mechanism rather than returning flag values. To make it easy to specify permitted and required signals, Larch/CLU contains some special syntactic sugar. A predicate of the form

normally Normal Predicate **except**
signals Signal Name_1 **when** Exception Guard_1
...
signals Signal Name_n **when** Exception Guard_n

is a shorthand for the predicate

(**returns** | **signals** Signal Name_1 | ... | **signals** Signal Name_n) &
(**returns** ⇒ (Normal Predicate &
¬(Exception Guard_1 | ... | Exception Guard_n))) &
(**signals** Signal Name_1 ⇒ Exception Guard_1) &
... &
(**signals** Signal Name_n ⇒ Exception Guard_n)

where **returns** and **signals** are Larch/CLU built-in predicates that deal with the possible ways for routines to terminate.

4. Notes on Two-Tiered Specifications

Larch can be used to write specifications that resemble operational specifications built on abstract models (e.g., [Hoare 72], [Berzins 79]). The Larch approach, however, differs in several important respects. The Shared Language is used to specify a theory, rather than a model, and the interface languages are built around predicate calculus rather than around an operational notation. One consequence of these differences is that Larch specifications are less prone to implementation bias.

It would be complicated to give semantic definitions of Larch/Pascal and Larch/CLU directly, because Pascal and CLU are complicated. Instead, we define the interface language semantics relative to the programming language semantics. This has two main advantages: we can be quite precise about what it means for an implementation to *satisfy* a specification, and we can provide a straightforward translation of a Larch interface language into predicate calculus.

The Larch Shared Language has mechanisms for building one specification from another (**assumes**, **includes**, and **imports**), and for inserting checkable redundancy into specifications (**constrains** and **converts**). The Larch interface languages do not have corresponding mechanisms. We wish to encourage a style of specification in which most of the programming-language-independent complexity is pushed into the traits, and interface specifications become almost trivial. We feel that specifiers are less likely to make serious mistakes in the simpler domain. Furthermore, it should be easier to provide machine support that will help them catch the mistakes that they do make. Finally, by encouraging them to put effort into traits, we increase the likelihood that parts of specifications will be reusable—not only for different specifications written in the same interface language, but also for specifications written in different interface languages.

The semantics of the Larch Shared Language is quite simple—except for some of the static error checking. This simplicity stems primarily from two decisions:

- All operators and sorts appearing in shared specifications are treated as "auxiliary." That is, operators and sorts are never implemented.
- Issues are not dealt with in the Shared Language if they must be dealt with in the interface languages.

As a result of the first decision, there is no mechanism to support the hiding of operators in the Shared Language. The hiding mechanisms of other specification languages 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. Since none of the operators appearing in a Shared Language specification are to be implemented, the introduction of a hiding mechanism would have no effect.

As a result of the second decision, there is no mechanism other than sort checking for restricting the domain of operators. Terms such as `eval(new, i)` in `TableSpec` are considered to be well-formed. Furthermore, no special "error" elements are introduced to represent the values of such terms. All preconditions and errors are handled in the interface languages. The Shared Language does include a mechanism for indicating that meanings of certain terms, such as `eval(new, i)`, have been intentionally left unconstrained. It may be desirable to check that the meaning of an interface specification does not depend on the meaning of **exempt** terms.

In this piece we present the Larch Shared Language before the Larch interface languages. This does not mean that traits are always written before the interface specifications that are based on them. In practice, we usually start by writing a trait, but we often go back and amend traits as we write interface specifications. In particular, we frequently add operators that enable us to write our predicates more concisely.

Piece II

The Larch Shared Language

1. Simple Algebraic Specifications

Most of the constructs in the Larch Shared Language are designed to assist in structuring specifications, for both reading and writing. The *trait* is our basic module of specification. Recall our specification for tables that store values in indexed places:

```
TableSpec: trait
    introduces
        new: → Table
        add: Table, Index, Val → Table
        #∈#: Index, Table → Bool
        eval: Table, Index → Val
        isEmpty: Table → Bool
        size: Table → Card
    constrains new, add, ∈, eval, isEmpty, size so that
        for all [ind, ind1: Index, val: Val, t: Table ]
            eval(add(t, ind, val), ind1) = if ind = ind1 then val else eval(t, ind1)
            ind ∈ new = false
            ind ∈ add(t, ind1, val) = (ind = ind1) | (ind ∈ t)
            size(new) = 0
            size(add(t, ind, val)) = if ind ∈ t then size(t) else size(t) + 1
            isEmpty(t) = (size(t) = 0)
```

This is similar to a conventional algebraic specification. The part of the specification following **introduces** declares a set of *operators* (function identifiers), each with its *signature* (the *sorts* of its domain and range). These 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 the specification constrains the operators by writing equations that relate sort-correct terms containing them.

There are two things (aside from syntactic amenities) that distinguish this specification from a specification written in our earlier algebraic specification languages:

- A name, TableSpec, is associated with the trait itself.
- The axioms are preceded by a **constrains** list.

The name of a trait is logically unrelated to any of the names appearing within it. In particular, we do not use sort identifiers to name units of specification. A trait need not correspond to a single abstract data type (ADT), and often does not.

The **constrains** list contains all of the operators that the immediately following axioms are intended to constrain. It is the responsibility of a specification checker to ensure that the specification conforms to this intent. The constrained operators will generally be a proper subset of the operators appearing in the axioms. In this example the **constrains** list informs us that the axioms are not to put any constraints on the properties of **if then else**, `false`, `0`, `1`, +, |, and =, despite their occurrence in the axioms. The judicious use of constrains lists is an important step in modularizing specifications.

We associate a theory with every trait. A theory is a set of well-formed formulas (wff's) of typed first-order predicate calculus with equations as atomic formulas.

The theory, call it Th, associated with a trait written in the Larch Shared Language is defined by:

- *Axioms:* Each equation, universally quantified by the variable declarations of the containing constrains clause, is in Th.
- *Inequation:* ¬(`true` = `false`) is in Th. All other inequations in Th are derivable from this one and the meaning of =.
- *First-order predicate calculus with equality:* Th contains the axioms of conventional typed first-order predicate calculus with equality and is closed under its rules of inference.

The equations and inequations in Th are derivable from the presence of axioms in the trait—never from their absence. It is important to prove theorems about specifications before they are complete, without worrying that adding new operators and equations will later invalidate some of them.

2. Getting Richer Theories

While the relatively small theory described above is often a useful one to associate with a set of axioms, there are times when a larger theory is needed, e.g., when specifying an abstract data type. **Generated by** and **partitioned by** give different ways of specifying larger theories.

Section 1 does not include an induction schema. Such a schema is not appropriate until the set of generators for a sort is complete. Saying that a sort is **generated by** a set of operators adds an inductive rule of inference. Intuitively, it asserts that the set contains sufficient operators to generate all values of the sort. For example, the natural numbers are **generated by** `0` and `successor` and the integers are **generated by** `0`, `successor`, and `predecessor`.

The clause `Table` **generated by** [`new`, `add`] can be used to derive theorems such as

$$\forall t\text{: Table } [\ (t = \text{new})\ |\ (\exists ind\text{: Index } [\ ind \in t\ ])\ ]$$

that would otherwise not be in the theory.

The rules of Section 1 allow equations to be derived by equational substitution, but not by the absence of inequations, since we do not want the addition of more equations to remove anything from the theory of a trait. Saying that sort S is **partitioned by** a set of operators, Ops, asserts that if two terms of sort S are unequal, a difference can be observed using an operator in Ops. Therefore, they must be equal if they cannot be distinguished using any of the operators in Ops. This adds new equations to the theory associated with a trait, thus reducing the number of equivalence classes in the equality relation.

The clause `Table` **partitioned by** [$\in$, `eval`] can be used to derive theorems such as

$$\text{add}(\text{add}(t,\ ind,\ v),\ ind_1,\ v) = \text{add}(\text{add}(t,\ ind_1,\ v),\ ind,\ v)$$

that would otherwise not be in the theory.

3. Combining Independent Traits

TableSpec contains a number of totally unconstrained operators, e.g., `false` and +. Such traits are not very useful. A straightforward thing to do is to augment the specification with additional clauses dealing with these operators. One way to do this is by trait *importation*. We might add to trait `TableSpec`:

imports `Cardinal`, `Boolean`

The theory associated with the importing trait is the theory associated with the union of all of the **introduces** and **constrains** clauses of the trait body and the imported traits.

Importation is used both to structure specifications to make them easier to read and to introduce extra checking. Operators appearing in imported traits may not be constrained by either the importing trait or by any other imported trait. This guarantees that imported traits don't "interfere" with one another in unexpected ways. I.e., it guarantees that the theory associated with a trait is a *conservative extension* of the theory associated with each of its imported traits. (Theory Th1 is a conservative extension of theory Th2 if the set of Th1's wffs that are in the language of Th2 is exactly Th2.) The operators of each imported trait can therefore be fully understood independently of the context into which the trait is imported.

As a syntactic amenity, trait `Boolean` is automatically imported into all other traits.

4. Combining Interacting Traits

While the modularity imposed by importation is often helpful, it can sometimes be too restrictive. 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. Trait *inclusion* involves the same union of clauses as trait importation, but allows the included operators to be further constrained. Consider, for example:

```
Reflexive: trait
    introduces # ⓡ #: T, T → Bool
    constrains ⓡ so that for all [ t: T ]
        t ⓡ t = true
Symmetric: trait
    introduces # ⓡ #: T, T → Bool
    constrains ⓡ so that for all [ t1, t2: T ]
        t1 ⓡ t2 = t2 ⓡ t1
Transitive: trait
    introduces # ⓡ #: T, T → Bool
    constrains ⓡ so that for all [ t1, t2, t3: T ]
        (((t1 ⓡ t2) & (t2 ⓡ t3))) ⇒ (t1 ⓡ t3)) = true
Equivalence: trait
    includes Reflexive, Symmetric, Transitive
```

Equivalence has the same associated theory as the less structured trait

```
Equivalence1: trait
    introduces # ⓡ #: T, T → Bool
    constrains ⓡ so that for all [ t1, t2, t3: T ]
        t1 ⓡ t1 = true
        t1 ⓡ t2 = t2 ⓡ t1
        (((t1 ⓡ t2) & (t2 ⓡ t3)) ⇒ (t1 ⓡ t3)) = true
```

Any legal trait importation may be replaced by trait inclusion without either making the trait illegal or changing the associated theory. However, such a replacement sacrifices the checking that ensures that the imported traits may be understood independently of the context in which they are used. We use importation when we can incorporate a theory unchanged, inclusion when we cannot.

5. Renaming

The specification of Equivalence in the previous section relied heavily on the coincidental use of the operator ⓡ and the sort identifier T in three separate traits. In the absence of such happy coincidences, renaming can force names to coincide, keep them from coinciding, or simply replace them with more suitable names.

The phrase

    Tr with [id_1 for id_2]

stands for the trait Tr with every occurrence of id_2 (which must be either a sort or operator identifier) replaced by id_1. Notice that if id_2 is a sort identifier this renaming may change the signatures associated with some operators.

If TableSpec contains the **generated by** and **partitioned by** of section 2, the specification

```
ArraySpec: trait
    imports IntegerSpec
    includes TableSpec with [ defined for # ∈ #, assign for add, read for eval,
            Array for Table, Integer for Index ]
```

stands for

```
ArraySpec: trait
    imports IntegerSpec
    introduces
        new: → Array
        assign: Array, Integer, Val → Array
        defined: Integer, Array → Bool
        read: Array, Integer → Val
        isEmpty: Array → Bool
        size: Table → Card
    constrains new, assign, defined, read, isEmpty so that
        Array generated by [ new, assign ]
        Array partitioned by [ defined, read ]
        for all [ind, ind1: Integer, val: Val, t: Array ]
            read(assign(t, ind, val), ind1) =
                if ind = ind1 then val else read(t, ind1)
            defined(ind, new) = false
            defined(ind1, assign(t, ind, val)) = ((ind = ind1) | defined(ind1, t))
```

```
size(new) = 0
size(add(t, ind, val)) =
    if defined(ind, t) then size(t) else size(t) + 1
isEmpty(t) = (size(t) = 0)
```

It is important to distinguish between the history of a specification (how it was constructed) and the structure presented to a reader. A reader familiar with `TableSpec` might prefer to read the first version of ArraySpec; others might find it distracting to have to understand the more general structure before understanding ArraySpec.

6. Recording Assumptions

We often construct fairly general specifications that we anticipate will later be specialized in a variety of ways. Consider, for example,

```
BagSpec: trait
    introduces
        {}: → Bag
        insert: Bag, Elem → Bag
        delete: Bag, Elem → Bag
        # ∈ #: Bag, Elem → Bool
    constrains {}, insert, delete, ∈ so that
        Bag generated by [ {}, insert ]
        Bag partitioned by [ delete, ∈ ]
        for all [ b: Bag, e, e1: Elem ]
            e ∈ {} = false
            e ∈ insert(b, e1) = (e = e1) | (e ∈ b)
            delete({}, e) = {}
            delete(insert(b, e), e1) =
                if e = e1 then b else insert(delete(b, e1), e)
```

We might specialize this to IntBag by renaming Elem to Integer and including it in a trait in which operators dealing with Integer are specified, e.g.,

```
IntBag: trait
    imports IntegerSpec
    includes BagSpec with [ Integer for Elem ]
```

The interactions between BagSpec and IntegerSpec are very limited. Nothing in BagSpec makes any assumptions about the meaning of the operators (other than =) that occur in IntegerSpec, e.g., 0, +, and <. Consider, however, extending BagSpec to BagSpec1 by adding an operator rangeCount,

```
BagSpec1: trait
    imports BagSpec, Cardinal
    introduces
        rangeCount: Bag, Elem, Elem → Integer
        # < #: Elem, Elem → Bool
    constrains rangeCount so that for all [ e1, e2, e3: Elem, b: Bag ]
        rangeCount({}, e1, e2) = 0
        rangeCount(insert(b, e3), e1, e2)) =
            rangeCount(b, e1, e2) + (if (e1 < e3)&(e3 < e2) then 1 else 0)
```

BagSpec1 makes no assumptions about the properties of the < operator. Suppose, however, that this is not what we intend. We might have definite ideas about the properties that < must have in any specialization, e.g., that it should define a total ordering. We specify such a restriction with an assumption:

```
BagSpec2: trait
    assumes Ordered with [ Elem for T ]
    imports BagSpec, Cardinal
    introduces
        rangeCount: Bag, Elem, Elem → Integer
    constrains rangeCount so that for all [ e1, e2, e3: Elem, b: Bag ]
        rangeCount({}, e1, e2) = 0
        rangeCount(insert(b, e3), e1, e2) =
            rangeCount(b, e1, e2) + (if (e1 < e3)&(e3 < e2) then 1 else 0)
```

The theory associated with BagSpec2, is the same as if

```
Ordered with [ Elem for T ]
```

had been included. This could be used to derive various properties of BagSpec2, e.g., that rangeCount is monotonic in its last argument.

Whenever BagSpec2 is imported or included in another trait, however, the assumption will have to be discharged. In

```
IntBag1: trait
    includes BagSpec2 with [ Integer for Elem ]
    imports IntegerSpec
```

this would amount to showing that the (renamed) theory associated with Ordered is a subset of the theory associated with IntegerSpec. Often, the assumptions of a trait are used to discharge the assumptions of traits it imports or includes.

7. Stating Intended Consequences

We have now looked at those parts of the Larch Shared Language that determine the theory associated with a legal trait. That subset of the language contains some checkable redundancy; e.g., assumptions are checked when a trait is included or imported, and **constrains** lists are checked against the axioms associated with them. We now turn to a part of the language whose only purpose is to introduce checkable redundancy, in the form of assertions about the theory associated with a trait.

There are two kinds of consequence assertions:

- That the theory associated with a trait contains another theory.
- That the theory associated with a trait adequately defines a set of operators in terms of other operators.

The first kind of assertion is made using **implies**. Consider, for example, adding to `BagSpec2`,

implies for all [*b*: `Bag`, e_1, e_2, e_3: `Elem`]
    $(e_2 < e_3) \Rightarrow (\texttt{rangeCount}(b, e_1, e_2) \leq \texttt{rangeCount}(b, e_1, e_3))$

Implies can be used to indicate intended consequences of a specification, both for checking and to increase the reader's insight. The theory to be implied can be specified using the full power of the language, e.g., by using **generated by** and **partitioned by**, or by referring to traits defined elsewhere.

The second kind of assertion is made using **converts** [Ops]. **Converts** is used to say that the specification adequately defines a collection of operators, i.e., that each term that contains no variables of any sort appearing in a **generated by** clause is provably equal to a term that does not contain any of the operators in Ops. A common problem with axiomatic systems is deciding whether there are enough axioms. **Converts** provides a way of making a checkable statement about the adequacy of a set of axioms. Consider, for example, adding to `TableSpec`:

converts [`isEmpty`]

This says that terms such as `isEmpty(new)` or `isEmpty(add(new,` *ind*, *val*`))`, are provably equal to terms that do not contain `isEmpty`.

Now consider adding to `TableSpec` the stronger assertion:

converts [`isEmpty, eval`]

Terms containing subterms of the form `eval(new,` *ind*) are not convertible to terms that do not contain `eval`, so an error message of the form

`eval(new,` *ind*) not convertible

would be generated. This incompleteness could be resolved by adding another axiom, for example

`eval(new,` *ind*) = `errorVal`

However, this requires recording a decision that might not be appropriate in such a trait, since it relies on the existence of an `errorVal` operator for sort `Val`. We therefore provide an **exempts** clause to indicate that the unconvertibility of certain terms is acceptable. If `TableSpec` were modifed to include

exempts for all [*ind:* `Index`] `eval(new,` *ind*)

the checking associated with the **converts** would now require that, for any term, t, which contains no variables of sort `Table`, the theory associated with `TableSpec` must contain either

- an equation, $t = t_1$, where t_1 has no occurrences of `isEmpty` or `eval`, or
- an equation $t' = t_1$, where t' is a subterm of t, and t_1 is an instantiation of `eval(new,` *ind*).

This checking ensures that each term containing operators in the **converts** list is either defined by the axioms (in terms of operators not in the list) or explicitly exempted.

8. IfThenElse and Equality

In our examples we made use of some apparently unconstrained operators: **if then else** and =, with a variety of signatures. The use of these operators leads to the implicit incorporation of the traits `IfThenElse` and `Equality`.

Whenever a term of the form **if** b **then** t_1 **else** t_2 occurs in a trait we replace the mixfix symbol **if then else** by the prefix symbol `ifThenElse`. If t_1 and t_2 are of the same sort, `T1`, we also import the trait

`IfThenElse` **with** [`T1` **for** `T`]

into the enclosing trait.

Whenever a term of the form $t_1 = t_2$ occurs in a trait, if t_1 and t_2 are of the same sort, `T1`, we append the trait

`Equality` **with** [`T1` **for** `T`]

to the consequences of the enclosing trait. These traits are defined in Piece III.

The operators `ifThenElse` and = are examples of *operator overloading*. In the Larch Shared Language, every operator is made up of an identifier or operator symbol and a signature. If the signature is deducible from context, it need not be written. This is why signatures appear only in the **introduces** clauses of the examples in this paper.

9. Further Examples

The following series of examples is adapted from Piece IV. Several of the examples have already been discussed in Piece I, section 2. We repeat them here to illustrate the coordinated use of the facilities introduced above, to introduce some syntactic sugar, and to serve as the basis for the definition of a generic operator at the end of the section.

The trait `Container` abstracts the common properties of those data structures that contain elements, e.g., sets, multisets, queues, and stacks. We have found it useful both as a starting point for specifications of many kinds of containers, and as an assumption when defining generic operators.

The **generated by** clause in `Container` indicates that each term that does not contain any variables of sort `C` is equal to some term in which `new` and `insert` are the only operators with range `C`. This assertion remains even if Container is included in another trait that introduces additional operators with range `C`. This means that any theorems proved by induction over `new` and `insert` will remain valid.

```
Container: trait
    introduces
        new: → C
        insert: C, E → C
    constrains C so that C generated by [ new, insert ]
```

The trait `IsEmpty` builds on `Container` by assuming it. It constrains the `new` and `insert` operators that it inherits from `Container`, as well as the operator that it introduces, `isEmpty`. The **converts** clause adds nothing to the theory of the trait. It adds checkable redundancy to the specification by indicating that this trait is intended to contain enough axioms to define `isEmpty`.

The two explicit axioms do not appear to be equations. This is because we have used a syntactic sugar that interprets single terms of sort `Bool` as equations by appending "= `true`".

```
IsEmpty: trait
    assumes Container
    introduces isEmpty: C → Bool
    constrains isEmpty, new, insert so that for all [ c: C, e: E ]
        isEmpty(new)
        ¬isEmpty(insert(c, e))
    implies converts [ isEmpty ]
```

Next and Rest also assume Container. Like **converts**, the **exempts** clauses are concerned with checking, and add nothing to the theory. They indicate that the lack of equations for next(new) and rest(new) is intentional.

```
Next: trait
    assumes Container
    introduces next: C → E
    constrains next, insert so that for all [ e: E ]
        next(insert(new, e)) = e
    exempts next(new)
```

```
Rest: trait
    assumes Container
    introduces rest: C → C
    constrains rest, insert so that for all [ e: E ]
        rest(insert(new, e)) = new
    exempts rest(new)
```

Enumerable augments Container by combining it with IsEmpty, Next, and Rest. The **includes** clause indicates that Enumerable is intended to inherit their operators and axioms and to further constrain the operators. The assumption of Container by the traits Next, Rest and IsEmpty is discharged in Enumerable by the explicit inclusion of Container.

The **partitioned by** clause indicates that next, rest, and isEmpty form a complete set of observer operators for sort C. This means that, for any terms t_1 and t_2, if the equalities next(t_1) = next(t_2), rest(t_1) = rest(t_2), and isEmpty(t_1) = isEmpty(t_2) all hold, then we may conclude that $t_1 = t_2$.

```
Enumerable: trait
    includes Container, Next, Rest, IsEmpty
    constrains C so that C partitioned by [ next, rest, isEmpty ]
```

PriorityQueue specializes Enumerable by further constraining next, rest, and insert. Sufficient axioms are given to convert next and rest. The axioms that convert isEmpty are inherited from the trait Enumerable, which inherited them from the trait IsEmpty.

The **with** clause indicates that the **assumed** trait is TotalOrder with the sort E substituted for the sort T throughout its text.

```
PriorityQueue: trait
    assumes TotalOrder with [ E for T ]
    includes Enumerable
    constrains next, rest, insert so that for all [ q: C, e: E ]
        next(insert(q, e)) =
            if isEmpty(q) then e
            else if next(q) ≤ e then next(q) else e
        rest(insert(q, e)) =
            if isEmpty(q) then new
            else if next(q) ≤ e then insert(rest(q), e) else q
    implies converts [ next, rest, isEmpty ]
```

Unlike the preceding traits in this section, PriorityQueue corresponds naturally to an abstract data type. In such a trait there will generally be a *distinguished sort* corresponding to the "type of interest" of [Guttag 75] or "data sort" of [Burstall and Goguen 81]. In such traits, it is usually possible to partition the operators whose range is the distinguished sort into *generators*, those operators which the sort is **generated by**, and *extensions*, which can be converted into generators. Operators whose domain includes the distinguished sort and whose range is some other sort are called *observers*. Observers are usually convertible, and the sort is usually **partitioned by** one or more subsets of the observers and extensions.

For example, in PriorityQueue, C is the distinguished sort, new and insert are generators, rest is an extension, and next and isEmpty are observers.

A good heuristic for generating enough equations to adequately define an abstract data type is to write one equation for each observer or extension applied to each generator. For PriorityQueue, this rule suggests axioms for rest(new), next(new), isEmpty(new), rest(insert(q, e)), next(insert(q, e)), and isEmpty(insert(q, e)). Note that the trait contains explicit equations for two of the six, and inherits equations for two more from IsEmpty. The remaining two, rest(new) and next(new), are **exempted** in Rest and Next.

The two remaining traits in this section specify generic operators. We assume `Enumerable` to ensure that these traits are used to define operators only on containers for which it is possible to enumerate the contained elements. (To understand why we assume `Enumerable` rather than `Container`, imagine defining `extOp` for a `MultiSet`.)

The **exempts** indicates that we do not intend to fully define the meaning of applying `extOp` to containers of unequal size. Notice that `elemOp` is totally unconstrained in this trait. This prevents us from having many interesting implications to state at this stage.

```
PairwiseExtension: trait
    assumes Enumerable
    introduces
        elemOp: E, E → E
        extOp: C, C → C
    constrains extOp so that for all [ c1, c2: C, e1, e2: E ]
        extOp(new, new) = new
        extOp(insert(c1, e1), insert(c2, e2)) =
            insert(extOp(c1, c2), elemOp(e1, e2))
    implies converts [ extOp ]
    exempts for all [ c: C, e: E ]
        extOp(new, insert(c, e)),
        extOp(insert(c, e), new)
```

Now we specialize `PairwiseExtension` by binding `elemOp` to + over `Cardinals`:

```
PairwisePlus: trait
    assumes Enumerable
    imports Cardinal
    includes PairwiseExtension with
            [ #+# for elemOp, #+# for extOp, Card for E ]
    implies Commutative with [ #+# for ∘, C for T ]
```

Trait `Commutative` appears in Piece IV. The validity of the implication that + (of sort `C`) is commutative stems from the replacement of `elemOp` by + (of sort `Card`), whose constraints (in trait `Cardinal`) imply its commutativity.

10. Discussion

We felt that it was important to carry the design of the Larch Shared Language through to the smallest details. This ensured that we did not overlook things that would turn out to be less trivial than they appeared. It allowed us to complete and check a fair number of examples. Finally, it was a necessary preliminary to the development of the support tools that we envision for Larch. The language embodies a large number of decisions, some of them more fundamental than others.

Among the less fundamental decisions are those dealing with syntax. We tried to make the surface syntax of the Shared Language comprehensible to readers of specifications, even at the expense of requiring quite a lot of punctuation (e.g., many lengthy reserved words). However, there is still room for experimentation and improvement here. It might make sense to adopt a more terse basic notation, and provide a variety of reading aids (e.g., prettyprinters, cross-reference tools) in a full-blown system.

The rest of this section touches on more fundamental decisions. These decisions may be wrong, but it would probably not be easy to change any of them without significantly affecting the character of the language.

A key assumption underlying our design was that specifications should be constructed and reasoned about incrementally. This led us to a design that ensures that adding things to a trait never removes formulas from its associated theory. The desire to maintain this monotonicity property led us to construe the equations of a trait as denoting a first-order theory. Had we chosen to take the theory associated with either the initial or final interpretation of a set of equations (as in [ADJ 78] and [Wand 79]), the monotonicity property would have been lost.

While we felt that many traits would correspond to complete abstract data types, we felt that many would not. This led us to introduce **generated by** and **partitioned by** as independent constructs. **Generated by** is used to close a set of constructors of a sort, and **partitioned by** to close a set of observers. Separating these constructs affords the specifier considerable flexibility.

Great flexibility is also afforded by the freedom to substitute, in a **with** list, for any operator or sort identifier in a trait. In effect, all such identifiers in a trait are formals. In an earlier version of the Larch Shared Language we had explicit lambda abstraction. We discovered, however, that our initial assumptions about which names to make parameters were often incorrect. In particular, we discovered that often we wished to substitute for a name that we had failed to make a parameter. On the other hand, we frequently used the same identifier for the actual as the formal, because in specific instances we did not need to use all the potential parameters.

Another important aspect of names in the Larch Shared Language is that operator names are qualified by a signature rather than by a single sort or by a trait. This is in contrast to many programming languages, e.g., CLU. This decision was forced upon us by our desire to make heavy use of overloading in specifications.

Reading specifications is an important activity, and what one sees when reading a specification is a syntactic object, i.e., a trait, rather than the theory. For this reason, we chose to use syntactic transformations to define the mechanisms for combining Larch Shared Language specifications. 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. In an earlier version of the Larch Shared Language [Guttag and Horning 83b], we had one mechanism that violated this property, **without**.

We devoted a great deal of attention to mechanisms for introducing checkable redundancy into specifications. **Assumes**, **imports**, and **includes** differ only in the checking associated with each. **Constrains** lists and the consequences section have no effect on the theory associated with a trait. They exist only to supply checkable redundancy. We chose to make the introduction of redundancy relatively fine-grained. Thus, for example, we have **constrains** lists of operators rather than lists of "protected" sorts.

The introduction of mechanisms to facilitate checking was not without some cost. The Larch Shared Language would be considerably smaller without them. Furthermore, experience indicates that it takes people roughly as long to learn those parts of the language involved with checking as it does to learn the part required to generate theories.

In contrast to our emphasis on syntactic mechanisms for building traits, we included a number of semantic constraints on the legality of traits, which were chosen to detect classes of errors that we expected to be common. A theorem prover will be the heart of any implementation of the Larch Shared Language. Most of the properties to be checked are undecidable. Thus the best that any checker can do is to answer "definitely OK," "definitely bad," or "too hard." We think that for most of the checks, the third answer will not occur too frequently. Although we don't yet have much experience to support this belief, we are encouraged by recent progress in the area of rewrite rule systems generally, and the Reve system specifically [Forgaard 84], [Lescanne 83].

In many respects, the Larch Shared Language is distinguished as much by what it doesn't include as by what it does.

The Shared Language provides no mechanism for "hiding" operators. The hiding mechanisms of other specification languages allow one to introduce 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. However, the operators appearing in a Shared Language specification are all auxiliary. Thus the introduction of a hiding mechanism would have no effect. Alternatively, we could say that the entire Shared Language tier is hidden.

There is no mechanism other than sort checking for restricting the domain of operators. Terms such as `eval(new,` *i*) are considered to be well-formed. Furthermore, no special "error" elements are introduced to represent the value of such terms. As discussed in the previous section, preconditions and errors are handled in the interface languages.

Similarly, nondeterminism is left to the interface languages. It is frequently useful to write incomplete specifications that admit distinct equivalence relations on terms (and non-isomorphic models). That is to say there are distinct terms that are neither provably equal nor provably unequal. However, it is always the case that for every term t, t = t. The whole mathematical basis of algebra and the Larch Shared Language depends on the ability to freely substitute "equals for equals." This property would be destroyed by the introduction of "nondeterministic functions."

Since our approach to specification frequently leads us to construct traits in which many things are left unconstrained, we do not include "completeness" among the properties that are required of a well-formed trait. Instead, we provide mechanisms (**converts** and **exempts**) that allow the specifier to state which completeness properties are to be checked. The choice will often depend on the intended interaction between a trait and the interface specifications that use it.

We have chosen not to use "higher-order" entities in the Larch Shared Language. Traits are simple textual objects. Their associated theories are first-order theories. We have completely sidestepped the subtle semantic problems associated with parameterized theories, theory parameters, and the like [Ehrig, *et al.* 80].

Piece III

The Larch Shared Language Reference Manual

Structure of the Manual

This piece is a self-contained reference manual for the Larch Shared Language. In it we give the syntax and static semantics of the Larch Shared Language. We also define how theories are associated with traits.

- Section 1 presents a grammar for the kernel subset of the Larch Shared Language.
- Section 2 defines the context sensitive checking and the theory associated with each specification written in the kernel subset.
- Section 3 extends the kernel subset by introducing mechanisms for specifying intended consequences of a specification written in the kernel subset.
- Sections 4-10 define successive extensions to the language. They extend the grammar to introduce additional aspects of the language and describe any additional context sensitive checking required. They also provide a translation from the newly extended language to the previously defined subset. The result of this translation is subject to the checking applicable to the target subset. The theory associated with any specification written in the full language is the same as the theory associated with its translation to the kernel subset.
- Section 11 describes additional checks, defined in terms of the theories associated with traits, that are associated with various language features. To be legal, a specification and each of the parts from which it is built must satisfy these checks in addition to the context sensitive checks described earlier.
- Section 12 collects the reference grammar for the entire language.

1. Kernel Language Syntax

<table>
<tr><td>trait</td><td>::=</td><td>traitId : trait traitBody</td></tr>
<tr><td>traitBody</td><td>::=</td><td>simpleTrait</td></tr>
<tr><td>simpleTrait</td><td>::=</td><td>{opPart} propPart*</td></tr>
<tr><td>opPart</td><td>::=</td><td>introduces opDcl*</td></tr>
<tr><td>opDcl</td><td>::=</td><td>opId : signature</td></tr>
<tr><td>signature</td><td>::=</td><td>domain → range</td></tr>
<tr><td>domain</td><td>::=</td><td>sortId*,</td></tr>
<tr><td>range</td><td>::=</td><td>sortId</td></tr>
<tr><td>propPart</td><td>::=</td><td>asserts props</td></tr>
<tr><td>props</td><td>::=</td><td>generators* partitions* axioms*</td></tr>
<tr><td>generators</td><td>::=</td><td>sortId generated bylist*,</td></tr>
<tr><td>partitions</td><td>::=</td><td>sortId partitioned bylist*,</td></tr>
<tr><td>bylist</td><td>::=</td><td>by [sortedOp*,]</td></tr>
<tr><td>sortedOp</td><td>::=</td><td>opDcl</td></tr>
<tr><td>axioms</td><td>::=</td><td>for all [varDcl*,] equation*</td></tr>
<tr><td>varDcl</td><td>::=</td><td>varId*, : sortId</td></tr>
<tr><td>equation</td><td>::=</td><td>term = term</td></tr>
<tr><td>term</td><td>::=</td><td>sortedOp { '(term*, ') } | varId</td></tr>
<tr><td>opId</td><td>::=</td><td>alphaNumeric+ | opForm</td></tr>
<tr><td>opForm</td><td>::=</td><td>{ # } opSym (# opSym)* { # }</td></tr>
<tr><td>opSym</td><td>::=</td><td>specialChar+ | . alphaNumeric+</td></tr>
<tr><td>traitId</td><td>::=</td><td>alphaNumeric+</td></tr>
<tr><td>sortId</td><td>::=</td><td>alphaNumeric+</td></tr>
<tr><td>varId</td><td>::=</td><td>alphaNumeric+</td></tr>
</table>

Comments start with % and terminate with end of line. They may appear after any token.

Syntactic conventions

<table>
<tr><td>|</td><td>alternative separator</td></tr>
<tr><td>{ e }</td><td>e is optional</td></tr>
<tr><td>e*</td><td>zero or more e's</td></tr>
<tr><td>e*,</td><td>zero or more e's, separated by commas</td></tr>
<tr><td>e+</td><td>one or more e's</td></tr>
<tr><td>alpha</td><td>alpha is a nonterminal symbol</td></tr>
<tr><td>alpha</td><td>alpha is a terminal symbol</td></tr>
<tr><td>'(')</td><td>parentheses as terminal symbols</td></tr>
<tr><td>(e)</td><td>parentheses for grouping syntactic expressions</td></tr>
</table>

2. Simple Traits

Context sensitive checking

simpleTrait:

- The sets of *varIds*, *sortIds* and *opIds* appearing in the *simpleTrait* must be disjoint.
- Each *sortId* and each *sortedOp* appearing anywhere in the *simpleTrait* must appear in its *opPart*.

opDcl:

- If the *opId* is an *opForm* it must have the same number of #'s as the number of occurrences of *sortIds* in the *signature*'s *domain*.

generators:

- The *range* of each *sortedOp* must be the *sortId* of the *generators*.
- At least one *sortedOp* in each *bylist* must have a *domain* in which the *sortId* of the *generators* does not occur.

partitions:

- The *domain* of each *sortedOp* must include the *sortId* of the *partitions*.
- The *range* of at least one *sortedOp* in each *bylist* must be different from the *sortId* of the *partitions*.

axioms:

- Each *varId* used in a *term* must appear in exactly one *varDcl*.
- No *varId* may occur more than once in [*varDcl**,].

equation:

- The sorts of both *terms* must be the same, where
  - The sort of a *term* of the form *sortedOp* { '(*term**, ') } is the *range* of the *sortedOp*.
  - The sort of a *term* of the form *varId* is the *sortId* of the *varDcl* in which the *varId* is declared.

term:

- In *sortedOp* { '(*term**, ') } the *domain* of the *sortedOp* must be the sequence of the sorts of the *terms* in *term**, .

Associated theory

We associate a theory with each *trait*. A theory is an inference-closed set of well-formed formulas (wffs) of typed first-order predicate calculus with equality. This section defines the theory associated with a *simpleTrait*.

We adopt the conventional meanings of the equality symbol (=), the propositional connectives (&, |, ¬, ⇒, ...), and the quantifiers (∀ and ∃). Since we use the same symbols to denote connectives as to denote the operators of the built-in traits `Boolean` and `Equality`, wffs containing unquantified terms can be ambiguous. However, since traits `Boolean` and `Equality` give the propositional connectives and = the same meanings as the corresponding predicate connectives, the ambiguity is harmless.

The theory, call it Th, associated with a *simpleTrait* is defined by:

- *Axioms:* Each *equation*, universally quantified by the *varDcls* of its containing *axioms*, is in Th.
- *Inequation:* ¬(`true:→Bool` = `false:→Bool`) is in Th.
- *First-order predicate calculus with equality:* Th contains the axioms of conventional typed first-order predicate calculus with equality and is closed under its rules of inference.
- *Induction:* If the *trait* has a *generators* with *sortId* S and a *bylist* **by** $[\mathrm{op}_1, \ldots, \mathrm{op}_n]$, and $P(s)$ is a wff with a free variable, s, of sort S, Th contains the wff

  $\forall[s\text{: S}]\ P(s)$

  if for each op_i in $[\mathrm{op}_1, \ldots, \mathrm{op}_n]$

  $Q_i \Rightarrow P(\mathrm{op}_i(x_1, \ldots, x_k))$ is in Th,

  where k is the arity of op_i,

  the x_j's are variables that do not appear free in P, and

  Q_i is the conjunction of $P(x_j)$, for each j such that the *j-th* argument of op_i is of sort S.

- *Reduction:* If the *trait* has a *partitions* with *sortId* S and a *bylist* **by** $[\mathrm{op}_1,\ldots,\mathrm{op}_n]$, Th contains the wff

  $\forall[s_1, s_2\text{: S}](Q \Rightarrow s_1 = s_2)$

  where Q is the conjunction, for each op_i in $[\mathrm{op}_1,\ldots,\mathrm{op}_n]$,

  and each j such that the *j-th* argument of op_i is of sort S of:

  $\forall[x_1\text{: } S_1,\ldots, x_k\text{: } S_k]\ (\mathrm{Subst}(\mathrm{op}_i, j, s_1) = \mathrm{Subst}(\mathrm{op}_i, j, s_2))$, where

  $S_1, \ldots, S_k$ is the *domain* of op_i, and

  $\mathrm{Subst}(\mathrm{op}, j, s)$ is $\mathrm{op}(x_1, \ldots, x_k)$ with s substituted for x_j.

3. Consequences and Exemptions

Exempts and *consequences* affect only the checking (see section 11) and do not affect the theory. We add to the grammar the productions:

trait	::=	*traitId* : **trait** *traitBody* {*consequences*} {*exempts*}
consequences	::=	**implies** *conseqProps* {*converts*}
conseqProps	::=	*props*
converts	::=	**converts** *conversion**,
conversion	::=	[*sortedOp**,]
exempts	::=	**exempts** *exemptTerms**
exemptTerms	::=	{ **for all** [*varDcl**,] } *term**,

Context sensitive checking

conseqProps:

- If the *props* of the *conseqProps* is appended to the *propPart* of the containing *trait*, the resulting *trait* must satisfy the checks of section 2.

exempts:

- Each *term* must satisfy the checks of section 2.

4. Constrains Clauses

Constrains clauses affect only the checking (see section 11), not the theory. We add to the grammar the productions:

propPart	::=	*constrains props*
constrains	::=	**constrains** (*sortId* | *sortedOp**,) **so that**

Translation

- Replace the *constrains* by **asserts**.

5. Implicit Signatures and Partial OpForms

In the kernel language each *sortedOp* is an *opDcl*. Here we relax this restriction to allow omitted and partial signatures and omitted #'s. We add to the grammar the production:

sortedOp ::= *opId* { → *range* }

Context sensitive checking

- There must be a unique mapping from occurrences of *sortedOps* to *opDcls* of the *traitBody* such that the translation described below produces a legal *traitBody* and for each *sortedOp*, *opDcl* pair:
  - The *opIds* match, i.e.,

    They are the same, or

    They are both *opForms* and the one in the *sortedOp* is the same as the one in the *opDcl* with all #'s removed.
  - If the *sortedOp* includes → *range*, it is the same as the *range* of the *opDcl*.

Translation

- The checking ensures that each occurrence of a *sortedOp* corresponds to a unique *opDcl*. The translation is simply to replace it by that *opDcl*.

6. Mixfix Operators

In the language presented thus far, all operators are treated as either nullary or prefix. Here we relax that restriction. We replace the grammar for *term* by:

term	::=	*secondary* | **if** *secondary* **then** *secondary* **else** *term*
secondary	::=	{ *opSym* } *primary* (*opSym primary*)* { *opSym* }
primary	::=	*sortedOp* { '(*term**, ') } | *varId* | '(*term* ')

Translation

equation:

- It is necessary to resolve the grammatical ambiguity between the = connective in *equations* and the = *opSym*. In any *equation* the first occurrence of = that is not bracketed by parentheses or within an if **then else** is the *equation* connective; the remainder are *opSyms*. Parentheses can be used to enforce any desired parsing.

term:

- Translate each *term* of the form **if** b **then** t_1 **else** t_2 into a *term* of the form `ifThenElse`(b, t_1, t_2).

secondary:

- Translate each *secondary* containing *opSyms* into a *primary* of the form

  opId '(*term**, '),

  where

  - *opId* is derived by replacing each *primary* in the *secondary* by #.
  - *term**, is the sequence of *primarys*.

primary:

- After the previous translations have been performed, remove the outer parentheses from *primarys* of the form '(*term* ').

7. Boolean Terms as Equations

It is convenient to use terms of sort `Bool` as *equations*. We add to the grammar the production:

equation ::= *term*

Context sensitive checking

- The *term* must be of sort `Bool`.

Translation

- Replace the *term* by the *equation*

  term = `true`

8. External References

We add to the kernel grammar the productions:

traitBody	::=	*externals simpleTrait*
externals	::=	{*assumes*} {*imports*} {*includes*}
assumes	::=	**assumes** *traitRef**,
imports	::=	**imports** *traitRef**,
includes	::=	**includes** *traitRef**,
traitRef	::=	*traitId*
conseqProps	::=	*traitRef**, *props*

Context sensitive checking

externals:

- Recursive *externals* are not permitted; i.e., the *traitId* of the containing *trait* may not appear in an *externals*, nor in any partial translation of a *traitRef* in its *externals*.

Translation

The translation of a *trait* is derived bottom-up; i.e., before a *trait* with *traitRefs* is translated, each of its *traitRefs* is replaced by the translation of the *trait* labeled by that *traitRef*'s *traitId*. Let T be a *trait* whose *simpleTrait* is S and let E consist of the translations of the *traitRef*'s in T's *externals*. The translation of T consists of:

- An *opPart* containing S's *opDcls* and E's *opDcls*.
- A *propPart** containing S's *propParts* and E's *propParts*.
- A *consequences* containing the *props* of
  - T's *conseqProps*.
  - the *propParts* of the translations of the *traitRefs* in T's *conseqProps*.
  - E's *consequences*.
- An *exempts* containing T's *exemptTerms* and E's *exemptTerms*.

9. Modifications

We add to the grammar the productions:

traitRef	::=	*traitId* {*renaming*}
renaming	::=	**with** [(*sortRename* | *opRename*)*,]
sortRename	::=	*sortId* **for** *oldSort*
oldSort	::=	*sortId*
opRename	::=	*opId* **for** *oldOp*
oldOp	::=	*sortedOp*

Context sensitive checking

traitRef:

- No *sortedOp* may occur more than once as an *oldOp*.
- No *sortId* may occur more than once as an *oldSort*.
- Each *oldSort* must appear in an *opDcl* in the translation of the *trait* labeled by the *traitId*.
- There must be a unique mapping from *oldOp*s to *opDcl*s of the translation of the *trait* labeled by the *traitId*, such that for each *oldOp*, *opDcl* pair:
  - The *opId*s match (see section 5),
  - If the *oldOp* includes a *domain*, it is the same as the *domain* of the *opDcl*.
  - If the *oldOp* includes → *range*, it is the same as the *range* of the *opDcl*.

Translation

- The translation of the *trait* labeled by the *traitId* of the *traitRef* is modified by applying first the *opRename*s, and then the *sortRename*s:
  - Simultaneously, for each *opRename*, replace the *opId* part of each occurrence of the *opDcl* to which the *oldOp* maps by the *opId* of the *opRename*.
  - Then, simultaneously, for each *sortRename*, replace each occurrence of its *oldSort* by its *sortId*.

10. Implicit Incorporation of Boolean, IfThenElse, and Equality

Three traits, `Boolean`, `IfThenElse`, and `Equality`, are implicitly incorporated into various other *traits* to assure uniform meanings for the operators they constrain.

Translation

- Append the *traitRef*

  `Boolean`

  to the *imports* of each *trait* except `Boolean`.

- Append the *traitRef*

  `IfThenElse` **with** [`T1` **for** `T`]

  to the *imports* of each *trait* containing a *term* of the form

  if b **then** t_1 **else** t_2

  in which t_1 and t_2 have the same sort, `T1`.

- Append the *traitRef*

  `Equality` **with** [`T1` **for** `T`]

  to the *traitRef** of the *conseqProps* of each *trait* (except `Equality`) containing a *term* of the form

  $t_1 = t_2$

  in which t_1 and t_2 have the same sort, `T1`.

Built-in traits

```
Boolean: trait
    introduces
        true: → Bool
        false: → Bool
        ¬ #: Bool → Bool
        # & #: Bool, Bool → Bool
        # | #: Bool, Bool → Bool
        # ⇒ #: Bool, Bool → Bool
        # ≡ #: Bool, Bool → Bool
    asserts Bool generated by [ true, false ]
        for all [ b: Bool ]
            ¬true = false
            ¬false = true
            (true & b) = b
            (false & b) = false
            (true | b) = true
            (false | b) = b
            (true ⇒ b) = b
            (false ⇒ b) = true
            (true ≡ b) = b
            (false ≡ b) = ¬b
            implies converts [ ¬, &, | , ⇒, ≡ ]

IfThenElse: trait
    introduces ifThenElse: Bool, T, T → T
    asserts for all [ t1, t2: T ]
        ifThenElse(true, t1, t2) = t1
        ifThenElse(false, t1, t2) = t2
    implies converts [ ifThenElse ]

Equality: trait
    introduces # = #: T, T → Bool
    asserts T partitioned by [ = ]
        for all [ x, y, z: T ]
            (x=x)
            (x=y) = (y=x)
            ((x=y) & (y=z)) ⇒ (x=z)
```

11. Semantic Checking

In addition to the syntactic constraints specified above, we require that each *trait* be logically consistent, discharge the assumptions of its external traits, be a conservative extension of its *imports*, be properly constraining, and imply its *consequences*.

Consistency

A *traitBody* is *consistent* if its associated theory does not contain the *equation*

`true:→Bool = false:→Bool`

Assumptions

Let A(T) be all of the *assumes* of the *traits* imported or included in T, and R(T) be the result of translating T after removing these *assumes*. A(T) is *discharged* by T if the theory associated with the translation of each *traitRef* of A(T) is a subset of the theory associated with R(T).

Imports

The theory associated with a *trait* must be a *conservative extension* of the theory associated with the translation of each *traitRef* in its *imports*; i.e., if *trait* T1 imports T2 and W is a *wff* containing only operators introduced in T2, W is in the theory associated with T1 if and only if it is in the theory associated with T2.

Constraints

A *propPart* is *properly-constraining* if it implies properties of only the operators in its *constrains*. The occurrence of a *sortId* in a *constrains* stands for the list of all *sortedOps* in the containing *trait*'s *opPart* whose *signatures* include that *sortId*.

Let T be a *trait* and P be the *propPart*

  constrains *sortedOp**, **so that** *props*.

P is properly-constraining in the *trait* consisting of T plus P if and only if each *wff* in the theory associated with T plus P is also in the theory associated with T or else contains a *sortedOp* listed in *sortedOp**.

Since the translation of a *traitRef* converts **constrains** to **asserts**, this check is performed only on traits in which **constrains** appears explicitly.

Consequences

A *trait implies* its *consequences* if the theory associated with its *conseqProps* is a subset of the theory associated with the *trait* and the [*sortedOp**,] in each **converts** is convertible. Convertibility is defined using the theory and *exempts* of a *trait*.

conseqProps:

- The theory associated with *conseqProps* must be a subset of the theory of the *trait* in which the *consequences* appears. The theory associated with a *conseqProps* is the theory associated with the *traitBody*

  includes *traitRef**,
  opPart
  asserts *props*

  where *traitRef**, and *props* form the *conseqProps*, and *opPart* is the *opPart* of the *trait* in which the *consequences* appears.

conversion:

- Let C be a *conversion*. For each *term*, t, that contains no variables of any sort appearing in a *generators* in the containing *trait*, the theory of the containing *trait* must either
  - contain an *equation* $t = t_1$, where t_1 contains no *sortedOp* appearing in C's *sortedOp**, or
  - contain an *equation* $t' = t_1$, where t' is a subterm of t, and t_1 is an instantiation of a *term* appearing in an *exempts* of the containing *trait*.

12. Reference Grammar for The Larch Shared Language

trait ::= *traitId* : **trait** *traitBody* {*consequences*} {*exempts*}
traitBody ::= *externals simpleTrait*
externals ::= {*assumes*} {*imports*} {*includes*}
assumes ::= **assumes** *traitRef**,
imports ::= **imports** *traitRef**,
includes ::= **includes** *traitRef**,
traitRef ::= *traitId* {*renaming*}
renaming ::= **with** [(*sortRename* | *opRename*)*,]
sortRename ::= *sortId* **for** *oldSort*
oldSort ::= *sortId*
opRename ::= *opId* **for** *oldOp*
oldOp ::= *sortedOp*
sortedOp ::= *opDcl* | *opId* { → *range* }
simpleTrait ::= {*opPart*} *propPart**
opPart ::= **introduces** *opDcl**
opDcl ::= *opId* : *signature*
signature ::= *domain* → *range*
domain ::= *sortId**,
range ::= *sortId*
propPart ::= (**asserts** | *constrains*) *props*
constrains ::= **constrains** (*sortId* | *sortedOp**,) **so that**
props ::= *generators** *partitions** *axioms**
generators ::= *sortId* **generated** *bylist**,
partitions ::= *sortId* **partitioned** *bylist**,
bylist ::= **by** [*sortedOp**,]
axioms ::= **for all** [*varDcl**,] *equation**
varDcl ::= *varId**, : *sortId*
equation ::= *term* { = *term* }
term ::= *secondary* | **if** *secondary* **then** *secondary* **else** *term*
secondary ::= { *opSym* } *primary* (*opSym primary*)* { *opSym* }
primary ::= *sortedOp* { '(*term**, ') } | *varId* | '(*term* ')
opId ::= *alphaNumeric*$^+$ | *opForm*
opForm ::= { # } *opSym* (# *opSym*)* { # }
opSym ::= *specialChar*$^+$ | . *alphaNumeric*$^+$
traitId ::= *alphaNumeric*$^+$
sortId ::= *alphaNumeric*$^+$
varId ::= *alphaNumeric*$^+$
consequences ::= **implies** *conseqProps* {*converts*}
conseqProps ::= *traitRef**, *props*
converts ::= **converts** *conversion**,
conversion ::= [*sortedOp**,]
exempts ::= **exempts** *exemptTerms**
exemptTerms ::= { **for all** [*varDcl**,] } *term**,

Piece IV

A Larch Shared Language Handbook

Preface

This handbook consists of a collection of traits written in the Larch Shared Language. It is intended to serve three purposes:

- Provide a set of illustrative examples that help people to understand the Larch Shared Language.
- Provide a set of components that can be directly incorporated into other specifications.
- Provide a set of models upon which other specifications can be based.

We have tried to isolate the smallest useful increments of specification that it might be reasonable to use in other specifications. In particular, we have tried to provide traits that will make it convenient to specify the weak assumptions that characterize many of the more widely applicable specifications, especially in Sections 7 and 8. The traits in these sections are smaller and more numerous than is typical in specifications written from scratch, which sometimes leads to a somewhat overstructured appearance.

In addition to traits that we expect to be directly incorporated in specifications, we have included a number of traits intended primarily as patterns. Section 9 contains several such traits. Specifiers are more likely to edit these traits than to **include** them, because they will need similar operators with different arities.

We have mostly stuck to familiar examples. Since they describe well-understood mathematical entities, many of the traits, e.g., Integer, are atypically complete. In general, we expect most specifications to supply constraints, rather than complete definitions. Section 14 is more typical in this respect.

The support tools envisioned for Larch are not yet available. Transcriptions of traits in this paper have been mechanically checked for some properties. Several errors were found and corrected as a result of this checking, but others may not have been detected and some additional transcription errors may have crept in. Thus these traits should be given the same sort of credence as carefully written programs that have not been checked by a compiler.

We would like to be able to present specifications with the clarity and rigor of a mathematics text, as advocated in [Abrial 80]. In particular, the formal text should be accompanied by a substantial amount of informal commentary. However, the present Handbook contains only the formal material, and corresponds more nearly to an appendix of "collected formulas" than to a text.

Conventions

- The identifier T is used to identify the only interesting sort in generic traits.
- The identifiers C and E are used for "containing" and "element" sorts.
- The infix symbol #∘# is used to denote a generic binary operator.
- The infix symbol #ⓡ# is used to denote a generic relational operator.
- An **asserts** clause is used rather than a **constrains** clause when **constrains** would supply no information (e.g., because there is only one operator).

1. Basic Properties of Single Operators

Associative: **trait**
  introduces #∘#: T, T → T
  asserts for all [x, y, z: T]
    $(x \circ y) \circ z = x \circ (y \circ z)$

Commutative: **trait**
  introduces #∘#: T, T → Range
  asserts for all [x, y: T]
    $x \circ y = y \circ x$

Idempotent: **trait**
  introduces op: T → T
  asserts for all [x: T]
    op(op(x)) = op(x)

Involutive: **trait**
  introduces op: T → T
    op(op(x)) = x

2. Basic Properties of Binary Relations

Relation: **trait**
  introduces #ⓡ#: T, T → Bool

TotalRelation: **trait**
  includes Relation
  asserts for all [x, y: T]
    $(x$ ⓡ $y) \mid (y$ ⓡ $x)$

Reflexive: **trait**
  includes Relation
  asserts for all [x: T]
    x ⓡ x

Irreflexive: **trait**
  includes Relation
  asserts for all [x: T]
    $\neg(x$ ⓡ $x)$

Transitive: **trait**
  includes Relation
  asserts for all [x, y, z: T]
    $((x$ ⓡ $y)$ & $(y$ ⓡ $z)) \Rightarrow (x$ ⓡ $z)$

ReflexiveTransitive: **trait**
  includes Reflexive, Transitive

Symmetric: **trait**
  includes Relation
  asserts for all [x, y: T]
    $(x$ ⓡ $y) = (y$ ⓡ $x)$
  implies Commutative **with** [ⓡ **for** ∘, Bool **for** Range]

Equivalence: **trait**
  includes ReflexiveTransitive **with** [.eq **for** ⓡ],
    Symmetric **with** [.eq **for** ⓡ]

3. Ordering Relations

```
PartialOrder: trait
    imports ReflexiveTransitive with [ ≤ for ⓡ ]

TotalOrder: trait
    includes PartialOrder, TotalRelation with [ ≤ for ⓡ ]

OrderEquivalence: trait
    assumes PartialOrder
    introduces #.eq#: T, T → Bool
    constrains .eq so that for all [ x, y: T ]
        (x .eq y) = (x ≤ y) & (y ≤ x)
    implies Equivalence
    converts [ .eq ]

OrderEquality: trait
    assumes PartialOrder
    includes Equality, OrderEquivalence with [ = for .eq ]

PartialOrderWithEquality: trait
    includes PartialOrder, OrderEquality

TotalOrderWithEquality: trait
    includes TotalOrder, OrderEquality

DerivedOrders: trait
    assumes PartialOrder
    introduces
        #<#: T, T → Bool
        #≥#: T, T → Bool
        #>#: T, T → Bool
    constrains < so that for all [ x, y: T ]
        (x < y) = ((x ≤ y) & (¬(y ≤ x)))
    constrains ≥ so that for all [ x, y: T ]
        (x ≥ y) = (y ≤ x)
    constrains > so that for all [ x, y: T ]
        (x > y) = (y < x)
    implies Transitive with [ < for ⓡ ],
        Transitive with [ > for ⓡ ],
        PartialOrder with [ ≥ for ≤ ]
    converts [ <, ≥, > ]
```

PartiallyOrdered: **trait**
  imports PartialOrderWithEquality
  includes DerivedOrders
  implies PartialOrderWithEquality **with** [$\geq$ **for** $\leq$]

Ordered: **trait**
  imports TotalOrderWithEquality
  includes DerivedOrders
  implies PartiallyOrdered, TotalOrderWithEquality **with** [$\geq$ **for** $\leq$]

4. Group Theory

LeftIdentity: **trait**
    introduces
        #∘#: T, T → T
        unit: → T
    asserts for all [x: T]
        unit ∘ x = x

RightIdentity: **trait**
    introduces
        #∘#: T, T → T
        unit: → T
    asserts for all [x: T]
        x ∘ unit = x

Identity: **trait includes** LeftIdentity, RightIdentity

LeftInverse: **trait**
    assumes LeftIdentity
    introduces inv: T → T
    asserts for all [x: T]
        inv(x) ∘ x = unit

RightInverse: **trait**
    assumes RightIdentity
    introduces inv: T → T
    asserts for all [x: T]
        x ∘ inv(x) = unit

Inverse: **trait**
    assumes Identity
    includes LeftInverse, RightInverse

Abelian: **trait imports** Commutative **with** [T **for** Range]

Semigroup: **trait includes** Associative, Equality

LeftMonoid: **trait includes** Semigroup, LeftIdentity

RightMonoid: **trait includes** Semigroup, RightIdentity

Monoid: **trait includes** LeftMonoid, RightMonoid

Group: **trait**
  includes LeftMonoid, LeftInverse
  implies RightMonoid, RightInverse, Involutive **with** [inv **for** op]

AbelianSemigroup: **trait includes** Abelian, Semigroup

AbelianMonoid: **trait**
  includes Abelian, LeftMonoid
  implies Monoid

AbelianGroup: **trait includes** Abelian, Group

Distributive: **trait**
  introduces
    #+#: T, T → T
    #*#: T, T → T
  asserts for all [x, y, z: T]
    $x * (y + z) = (x * y) + (x * z)$
    $(y + z) * x = (y * x) + (z * x)$

5. Simple Numeric Types

```
Ordinal: trait
    includes Ordered with [ Ord for T ]
    introduces
        first: → Ord
        succ: Ord → Ord
    asserts Ord generated by [ first, succ ]
        Ord partitioned by [ ≤ ]
        for all [ x, y: Ord ]
            first ≤ x
            ¬(succ(x) ≤ first)
            (succ(x) ≤ succ(y)) = (x ≤ y)
    converts [ =, ≤, <, ≥, > ]

Cardinal: trait
    imports Ordinal with [ 0 for first, Card for Ord ]
    introduces
        1: → Card
        #+#: Card, Card → Card
        #*#: Card, Card → Card
        #⊖#: Card, Card → Card
    constrains 1 so that 1 = succ(0)
    constrains + so that for all [ x, y: Card ]
        x + 0 = x
        x + succ(y) = succ(x + y)
    constrains * so that for all [ x, y: Card ]
        x * 0 = 0
        x * succ(y) = x + (x * y)
    constrains ⊖ so that for all [ x, y: Card ]
        0 ⊖ x = 0
        x ⊖ 0 = x
        succ(x) ⊖ succ(y) = x ⊖ y
    implies
        Cardinal2
        Card generated by [ 1, +, ⊖ ]
        Card partitioned by [ ≥ ], by [ = ], by [ < ], by [ > ]
        for all [ x, y: Card ] x ≤ y = ((x ⊖ y) = 0)
    converts [ 1, ⊖, +, *, =, ≤, ≥, <, > ]
```

```
Cardinal2: trait                % Alternate definition. Compare with Cardinal above.
    includes AbelianMonoid with [ + for ∘, 0 for unit, Card for T ],
        AbelianMonoid with [ * for ∘, 1 for unit, Card for T ],
        Distributive with [ Card for T ],
        Ordered with [ Card for T ]
    introduces
        #⊖#: Card, Card → Card
        succ: Card → Card
    asserts Card generated by [ 0, 1, + ]
        for all [ x, y: Card ]
            x < (x + 1)
            (x + y) ⊖ y = x
            0 ⊖ x = 0
            succ(x) = x + 1
    implies Cardinal
```

6. Simple Data Structures

```
Pair: trait
    introduces
        ⟨#, #⟩: T1, T2 → C
        #.first: C → T1
        #.second: C → T2
    asserts C generated by [ ⟨#, #⟩ ]
        C partitioned by [ .first, .second ]
        for all [ f: T1, s: T2 ]
            ⟨f, s⟩.first = f
            ⟨f, s⟩.second = s
    implies converts [ .first, .second ]

Triple: trait
    introduces
        ⟨#, #, #⟩: T1, T2, T3 → C
        #.first: C → T1
        #.second: C → T2
        #.third: C → T3
    asserts C generated by [ ⟨#, #, #⟩ ]
        C partitioned by [ .first, .second, .third ]
        for all [ f: T1, s: T2, t: T3 ]
            ⟨f, s, t⟩.first = f
            ⟨f, s, t⟩.second = s
            ⟨f, s, t⟩.third = t
    implies converts [ .first, .second, .third ]
```

FiniteMapping: **trait**

assumes Equality **with** [Index **for** T]

introduces

new: → C

bind: C, Index, E → C

#[#]: C, Index → E

defined: C, Index → Bool

asserts C **generated by** [new, bind]

C **partitioned by** [#[#], defined]

constrains C **so that**

for all [c: C, i, i_1: Index, e: E]

bind(c, i_1, e)[i] = **if** $i = i_1$ **then** e **else** c[i]

¬defined(new, i)

defined(bind(c, i_1, e), i) = ($i = i_1$) | defined(c, i)

implies converts [#[#], defined]

exempts for all [i: Index] new[i]

7. Container Properties

```
Container: trait
    introduces
        new: → C
        insert: C, E → C
    asserts C generated by [ new, insert ]

Singleton: trait
    assumes Container
    introduces singleton: E → C
    constrains singleton so that for all [ e: E ]
        singleton(e) = insert(new, e)
    implies converts [ singleton ]

IsEmpty: trait
    assumes Container
    introduces isEmpty: C → Bool
    asserts for all [ c: C, e: E ]
        isEmpty(new)
        ¬isEmpty(insert(c, e))
    implies converts [ isEmpty ]

Size: trait
    assumes Container
    imports Cardinal
    introduces size: C → Card
    constrains size so that
        size(new) = 0

AdditiveSize: trait
    assumes Container
    includes Size
    constrains size, insert so that for all [ c: C, e: E ]
        size(insert(c, e)) = size(c) + 1
    implies converts [ size ]
```

Join: **trait**
  assumes Container
  introduces #.join#: C, C → C
  constrains .join **so that for all** [c, c_1: C, e: E]
    c .join new = c
    c .join insert(c_1, e) = insert(c .join c_1, e)
  implies Associative **with** [.join **for** ∘]
  converts [.join]

ElementEquality: **trait imports** Equality **with** [E **for** T]

Member: **trait**
  assumes Container, ElementEquality
  introduces #∈#: E, C → Bool
  constrains ∈, insert **so that for all** [c: C, e, e_1: E]
    ¬(e ∈ new)
    e ∈ insert(c, e_1) = (e = e_1) | (e ∈ c)
  implies converts [∈]

ElemCount: **trait**
  assumes Container, ElementEquality
  imports Cardinal
  introduces count: C, E → Card
  constrains count, insert **so that for all** [e, e_1: E, c: C]
    count(new, e) = 0
    count(insert(c, e), e_1) = count(c, e) + (**if** e = e_1 **then 1 else 0**)
  implies converts [count]

Delete: **trait**
  assumes Container
  introduces delete: C, E → C
  constrains delete **so that for all** [e: E]
    delete(new, e) = new

Containment: **trait**
  assumes Container
  includes PartiallyOrdered **with**
    [⊆ **for** ≤, ⊇ **for** ≥, ⊂ **for** <, ⊃ **for** >, C **for** T]
  constrains C **so that for all** [e: E, c: C]
    c ⊆ insert(c, e)
  implies for all [c: C]
    new ⊆ c

```
Next: trait
    assumes Container
    introduces next: C → E
    constrains next, insert so that for all [ e: E ]
        next(insert(new, e)) = e
    exempts next(new)

Rest: trait
    assumes Container
    introduces rest: C → C
    constrains rest so that for all [ e: E ]
        rest(insert(new, e)) = new
    exempts rest(new)

Remainder: trait
    assumes Container, Rest
    imports Cardinal
    introduces remainder: C, Card → C
    constrains remainder so that for all [ c: C, i: Card ]
        remainder(c, 0) = c
        remainder(c, i + 1) = remainder(rest(c), i)
    implies converts [ remainder ]

Index: trait
    assumes Container, Next, Rest
    imports Cardinal
    introduces #[#]: C, Card → E
    constrains #[#] so that for all [ c: C, i: Card ]
        c[1] = next(c)
        c[(i + 1)] = rest(c)[i]
    implies converts [ #[#] ]
    exempts for all [ c: C ] c[0]
```

8. Container Classes

SetBasics: **trait**
    assumes ElementEquality, Container **with** [{} **for** new]
    includes Size **with** [{} **for** new], Member **with** [{} **for** new]
    introduces delete: C, E → C
    constrains C **so that**
        C **partitioned by** [∈]
        for all [s: C, e, e_1: E]
            size(insert(s, e)) = size(s) + (**if** $e \in s$ **then** 0 **else** 1)
            $e_1 \in$ delete(s, e) = ($e_1 \in s$) & ($\neg$($e = e_1$))
    implies Delete **with** [{} **for** new]
    converts [size, delete, ∈]

BagBasics: **trait**
    assumes ElementEquality, Container **with** [{} **for** new]
    imports AdditiveSize **with** [{} **for** new],
        ElemCount **with** [{} **for** new]
    includes Member **with** [{} **for** new]
    introduces delete: C, E → C
    constrains C **so that**
        C **partitioned by** [count]
        for all [b: C, e, e_1: E]
            count(delete(b, e), e_1) = count(b, e_1) − (**if** $e = e_1$ **then** 1 **else** 0)
    implies Delete **with** [{} **for** new]
    converts [size, delete, count, ∈]

CollectionExtensions: **trait**
    assumes ElementEquality, Container **with** [{} **for** new]
    imports IsEmpty **with** [{} **for** new],
        Singleton **with** [{} **for** new, {#} **for** singleton],
        Containment **with** [{} **for** new],
        Join **with** [{} **for** new, ∪ **for** .join]
    includes Equality **with** [C **for** T]
    implies converts [{#}, isEmpty, ∪]

SetIntersection: **trait**
  assumes SetBasics
  introduces #∩#: C, C → C
  constrains ∩ **so that for all** [s, s_1: C, e: E]
    $e \in (s \cap s_1) = (e \in s) \;\&\; (e \in s_1)$
  converts [∩]

Set: **trait**
  assumes ElementEquality
  imports SetBasics, SetIntersection
  includes CollectionExtensions
  implies Abelian **with** [∪ **for** ∘, C **for** T],
    Abelian **with** [∩ **for** ∘, C **for** T]
  converts [size, delete, ∈, ∩, ∪, {#}, isEmpty, =, ⊆, ⊇, ⊂, ⊃]

Bag: **trait**
  assumes ElementEquality
  imports BagBasics
  includes CollectionExtensions
  implies Abelian **with** [∪ **for** ∘, C **for** T]
  converts [size, delete, count, ∈, ∪, {#}, isEmpty, =, ⊆, ⊇, ⊂, ⊃]

Enumerable: **trait**
  imports IsEmpty, Next, Rest
  includes Container
  constrains C **so that** C **partitioned by** [next, rest, isEmpty]

Stack: **trait**
  includes Enumerable **with** [push **for** insert, top **for** next, pop **for** rest]
  constrains push, pop, top **so that for all** [*stk*: C, *e*: E]
    top(push(*stk*, *e*)) = *e*
    pop(push(*stk*, *e*)) = *stk*

Queue: **trait**
  includes Enumerable **with** [first **for** next]
  constrains first, rest, insert **so that for all** [*q*: C, *e*: E]
    first(insert(*q*, *e*)) = **if** isEmpty(*q*) **then** *e* **else** first(*q*)
    rest(insert(*q*, *e*)) = **if** isEmpty(*q*) **then** new **else** insert(rest(*q*), *e*)

Deqeue: **trait**

includes Stack **with** [insert **for** push, first **for** top, rest **for** pop],
Stack **with** [enter **for** push, last **for** top, prefix **for** pop]
constrains C **so that for all** [*c:* C, *e*, e_1*:* E]
insert(new, *e*) = enter(new, *e*)
insert(enter(*c*, *e*), e_1) = enter(insert(*c*, e_1), *e*)
implies Queue, Queue **with** [enter **for** insert, last **for** first, prefix **for** rest]
converts [insert, first, last, rest, prefix], [enter, first, last, rest, prefix]

Sequence: **trait**

imports Dequeue, AdditiveSize
includes Index **with** [first **for** next],
Join **with** [|| **for** .join]
implies C **partitioned by** [size, #[#]]

SubSequence: **trait**

imports Sequence
includes Remainder **with** [#[#...] **for** remainder],
Remainder **with** [#[...#] **for** remainder, prefix **for** rest]

PriorityQueue: **trait**

assumes TotalOrder **with** [E **for** T]
includes Enumerable
constrains next, rest, insert **so that for all** [*q:* C, *e:* E]
next(insert(*q*, *e*)) = **if** isEmpty(*q*) **then** *e*
else if next(*q*) ≤ *e* **then** next(*q*) **else** *e*
rest(insert(*q*, *e*)) = **if** isEmpty(*q*) **then** new
else if next(*q*) ≤ *e* **then** insert(rest(*q*), *e*) **else** *q*
implies converts [next, rest, isEmpty]

9. Generic Operators on Containers

```
CoerceContainer: trait
    assumes Container with [ DC for C ],
        Container with [ RC for C ]
    introduces coerce: DC → RC
    constrains coerce so that for all [ dc: DC, e: E ]
        coerce(new) = new
        coerce(insert(dc, e)) = insert(coerce(dc), e)
    implies converts [ coerce ]
```

```
Reduce: trait
    assumes Enumerable, RightIdentity with [ E for T ]
    introduces reduce: C → E
    constrains reduce so that for all [ c: C ]
        reduce(c) = if isEmpty(c) then unit else next(c) ∘ reduce(rest(c))
    implies converts [ reduce ]
```

```
SomePass: trait
    assumes Container
    introduces
        test: E, T → Bool
        somePass: C, T → Bool
    constrains somePass so that for all [ c: C, e: E, t: T ]
        ¬somePass(new, t)
        somePass(insert(c, e), t) = test(e, t) | somePass(c, t)
    implies converts [ somePass ]
```

```
AllPass: trait
    assumes Container
    introduces
        test: E, T → Bool
        allPass: C, T → Bool
    constrains allPass so that for all [ c: C, e: E, t: T ]
        allPass(new, t)
        allPass(insert(c, e), t) = test(e, t) & allPass(c, t)
    implies converts [ allPass ]
```

Sift: **trait**
    assumes Container
    introduces
        test: E, T → Bool
        sift: C, T → C
    constrains sift **so that for all** [c: C, e: E, t: T]
        sift(new, t) = new
        sift(insert(c, e), t) = **if** test(e, t) **then** insert(sift(c, t), e) **else** sift(c, t)
    implies converts [sift]

PairwiseExtension: **trait**
    assumes Enumerable
    introduces
        extOp: C, C → C
        elemOp: E, E → E
    constrains extOp **so that for all** [c_1, c_2: C, e_1, e_2: E]
        extOp(new, new) = new
        extOp(insert(c_1, e_1), insert(c_2, e_2)) =
            insert(extOp(c_1, c_2), elemOp(e_1, e_2))
    implies converts [extOp]
    exempts for all [c: C, e: E]
        extOp(new, insert(c, e)),
        extOp(insert(c, e), new)

PointwiseImage: **trait**
    assumes Container **with** [DC **for** C, DE **for** E],
        Container **with** [RC **for** C, RE **for** E]
    introduces
        extOp: DC → RC
        pointOp: DE → RE
    constrains extOp **so that for all** [dc: DC, de: DE]
        extOp(new) = new
        extOp(insert(dc, de)) = insert(extOp(dc), pointOp(de))
    implies converts [extOp]

10. Nonlinear Structures

BinaryTree: **trait**
    imports Cardinal
    introduces
        $\langle\#\rangle$: E → C
        $\langle\#, \#\rangle$: C, C → C
        #.left: C → C
        #.right: C → C
        size: C → Card
        isLeaf: C → Bool
        content: C → E
    constrains C **so that**
        C **generated by** [$\langle\#\rangle$, $\langle\#, \#\rangle$]
        C **partitioned by** [.left, .right, content, isLeaf]
        for all [*tl*, *tr*: C, *e*: E]
            $(\langle tl, tr\rangle)$.left = *tl*
            $(\langle tl, tr\rangle)$.right = *tr*
            size$(\langle e\rangle) = 1$
            size$(\langle tl, tr\rangle)$ = size(*tl*) + size(*tr*)
            isLeaf$(\langle e\rangle)$
            ¬isLeaf$(\langle tl, tr\rangle)$
            content$(\langle e\rangle) = e$
    implies for all [*t*: C] isLeaf(*t*) = (size(*t*) = 1)
    converts [.left, .right, size, isLeaf, content]
    exempts for all [*tl*, *tr*: C, *e*: E] $(\langle e\rangle)$.left, $(\langle e\rangle)$.right, content$(\langle tl, tr\rangle)$

```
BasicGraph: trait
    assumes Equality with [ Node for T ]
    imports Set with [ NodeSet for C, Node for E ],
        Pair with [ Edge for C, Node for T1, Node for T2 ]
    introduces
        empty: → Graph
        addNode: Graph, Node → Graph
        addEdge: Graph, Edge → Graph
        nodes: Graph → NodeSet
        adj: Node, Graph → NodeSet
    constrains Graph so that
        Graph generated by [ empty, addNode, addEdge ]
        Graph partitioned by [ nodes, adj ]
        for all [ g: Graph, e: Edge, n, n1: Node ]
            nodes(empty) = {}
            nodes(addNode(g, n)) = insert(nodes(g), n)
            nodes(addEdge(g, e)) = insert(insert(nodes(g), e.first), e.second)
            adj(n, empty) = {}
            adj(n, addNode(g, n1)) = adj(n, g)
            adj(n, addEdge(g, e)) =
                if n = (e.first) then insert(adj(n, g), e.second) else adj(n, g)
    implies converts [ nodes, adj ]

Connectivity: trait
    assumes Equality with [ Node for T ], BasicGraph
    introduces
        reach: NodeSet, Graph → NodeSet
        allReach: NodeSet, NodeSet, Graph → Bool
        connected: Graph → Bool
    constrains reach, allReach, connected so that
        for all [ g: Graph, e: Edge, ns, ns1: NodeSet, n: Node ]
            reach(ns, empty) = {}
            reach(ns, addNode(g, n)) = reach(ns, g)
            allReach({}, ns, g)
            allReach(insert(ns, n), ns1, g) =
                allReach(ns, ns1, g) & (ns1 ⊆ reach({n}, g))
            connected(g) = allReach(nodes(g), nodes(g), g)
    implies converts [ allReach, connected ]
```

```
Graph: trait
    assumes Equality with [ Node for T ]
    imports BasicGraph
    includes Connectivity,
        Connectivity with [ stronglyConnected for connected, pathReach for reach,
                    allPathReach for allReach ]
    constrains reach, allReach, connected so that
        for all [ g: Graph, e: Edge, ns: NodeSet ]
            reach(ns, addEdge(g, e)) =
                reach(ns, g) ∪
                    (if (e.first) ∈ ns
                    then insert(reach({(e.second)}, g), (e.second))
                    else if (e.second) ∈ ns
                    then insert(reach({(e.first)}, g), (e.first))
                    else {})
    constrains pathReach, allPathReach, stronglyConnected so that
        for all [ g: Graph, e: Edge, ns: NodeSet ]
            pathReach(ns, addEdge(g, e)) =
                pathReach(ns, g) ∪
                    (if (e.first) ∈ ns
                    then insert(pathReach({(e.second)}, g), (e.second))
                    else {})
    implies converts [ reach, allReach, connected, pathReach, allPathReach,
                stronglyConnected ]
```

11. Rings, Fields, and Numbers

Ring: **trait**
  includes AbelianGroup **with** [+ **for** $\circ$, 0 **for** unit, $-\#$ **for** inv],
    Semigroup **with** [$*$ **for** $\circ$],
    Distributive

RingWithUnit: **trait**
  includes Ring, Identity **with** [$*$ **for** $\circ$, 1 **for** unit]

InfixInverse: **trait**
  assumes Inverse
  introduces $\#\oslash\#$: T, T $\rightarrow$ T
  constrains $\#\oslash\#$ **so that for all** [x, y: T]
    $x \oslash y = x \circ \mathrm{inv}(y)$
  implies converts [$\#\oslash\#$]

Integer: **trait**
  includes RingWithUnit **with** [Int **for** T],
    Ordered **with** [Int **for** T],
    InfixInverse **with** [+ **for** $\circ$, $-\#$ **for** inv, $-$ **for** $\oslash$, Int **for** T]
  asserts Int **generated by** [1, +, $-\#$]
    for all [x: Int]
      $x < (x + 1)$
  converts [0, $*$, $\#-\#$, $=$, $\leq$, $\geq$, $<$, $>$]

Field: **trait**
  includes RingWithUnit
  introduces $\#^{-1}$: T $\rightarrow$ T
  constrains $*$, $^{-1}$ **so that for all** [x: T]
    $(x = 0) \mid ((x*(x^{-1})) = 1)$
  exempts 0^{-1}

Rational: **trait**

includes Field **with** [R **for** T],

Ordered **with** [R **for** T],

InfixInverse **with** [+ **for** ∘, −# **for** inv, − **for** ⊘, R **for** T]

InfixInverse **with** [∗ **for** ∘, $\#^{-1}$ **for** inv, / **for** ⊘, R **for** T]

asserts

R **generated by** [1, +, −#, $^{-1}$]

for all [x, y, z: R]

$0 < 1$

$((x + z) < (y + z)) = (x < y)$

$(x = 0) \mid ((0 < (x^{-1})) = (0 < x))$

implies converts [0, ∗, #−#, /, =, ≤, ≥, <, >]

12. Lattices

ExtremalBound: **trait**

assumes PartialOrder

includes AbelianSemigroup **with** [.glb **for** $\circ$]

constrains .glb **so that for all** [x, y, z: T]

$(x \text{ .glb } y) \leq x$

$((z \leq x) \ \& \ (z \leq y)) \Rightarrow (z \leq (x \text{ .glb } y))$

Semilattice: **trait**

includes PartiallyOrdered,

ExtremalBound,

ExtremalBound **with** [$\geq$ for $\leq$, .lub **for** .glb]

introduces $\perp$: $\rightarrow$ T

constrains $\perp$ **so that for all** [x: T]

$x \geq \perp$

implies AbelianMonoid **with** [$\perp$ **for** unit, .lub **for** $\circ$]

Lattice: trait

includes Semilattice

introduces $\top$: $\rightarrow$ T

constrains $\top$ **so that for all** [x: T]

$x \leq \top$

implies Lattice **with** [$\top$ **for** $\perp$, $\perp$ **for** $\top$, .glb **for** .lub, .lub **for** .glb,

$\geq$ **for** $\leq$, $\leq$ **for** $\geq$, $>$ **for** $<$, $<$ **for** $>$]

13. Enumerated Data Types

```
Enumerated: trait
    imports Ordinal
    includes Ordered
    introduces
        first: → T
        last: → T
        succ: T → T
        pred: T → T
        ord: T → Ord
    asserts T generated by [ first, succ ]
        T partitioned by [ ord ]
        for all [ x, y: T ]
            ord(first) = first
            ord(succ(x)) = if x = last then ord(last) else succ(ord(x))
            pred(succ(x)) = if x = last then pred(last) else x
            (x ≤ y) = (ord(x) ≤ ord(y))
    implies T generated by [ last, pred ]
        for all [ x: T ]
            succ(pred(x)) = if x = first then succ(pred(first)) else x
            first ≤ x
            x ≤ last
    converts [ =, ≤, ≥, <, > ]
```

Rainbow: **trait**

    includes Enumerated **with** [Color **for** T]

    introduces

        red: → Color

        orange: → Color

        yellow: → Color

        green: → Color

        blue: → Color

        violet: → Color

    asserts

        Color **generated by** [red, orange, yellow, green, blue, violet]

        first = red

        last = violet

        succ(red) = orange

        succ(orange) = yellow

        succ(yellow) = green

        succ(green) = blue

        succ(blue) = violet

    implies converts

        [pred, last, ord, =, ≤, ≥, <, >, red, orange, yellow, green, blue, violet],

        [succ, first, ord, =, ≤, ≥, <, >, red, orange, yellow, green, blue, violet]

14. Display Traits

% The following traits represent a fairly straightforward translation of the specifications
% in "Formal Specification as a Design Tool" [Guttag and Horning 80]. We have
% not attempted to improve the design presented there, merely to translate it into Larch.

```
Coordinate: trait introduces minus: Coordinate, Coordinate → Coordinate

Illumination: trait introduces combine: Illumination, Illumination → Illumination

Boundary: trait introduces apply: Boundary, Coordinate → Bool

Transform: trait introduces apply: Transformation, Coordinate → Coordinate

Displayable: trait
    introduces
        appearance: T, Coordinate → Illumination
        in: T, Coordinate → Bool

Picture: trait
    assumes Boundary, Transform, Illumination,
        Displayable with [ Contents for T ]
    includes Displayable with [ Picture for T ]
    introduces makePicture: Contents, Boundary, Transformation → Picture
    constrains Picture so that
        Picture generated by [ makePicture ]
        for all [ cn: Contents, b: Boundary, t: Transformation, cd: Coordinate ]
            appearance(makePicture(cn, b, t), cd) =
                appearance(cn, apply(t, cd))
            in(makePicture(cn, b, t), cd) = apply(b, cd)
    implies converts [ appearance: Picture, Coordinate → Illumination,
            in: Picture, Coordinate → Bool ]
```

Contents: **trait**
  assumes Coordinate, Illumination, Displayable **with** [Component **for** T]
  includes Displayable **with** [Contents **for** T]
  introduces
    empty: → Contents
    addComponent: Contents, Component, Coordinate → Contents
  constrains Contents **so that**
    Contents **generated by** [empty, addComponent]
    for all [*cn:* Contents, *cm:* Component, *cd*, cd_1: Coordinate]
      appearance(addComponent(*cn*, *cm*, cd_1, *cd*)) =
        if in(*cm*, minus(*cd*, cd_1))
        then (**if** in(*cn*, *cd*)
          then combine(appearance(*cm*, minus(*cd*, cd_1)),
            (*cn*, *cd*))
          else appearance(*cm*, minus(*cd*, cd_1)))
        else appearance(*cn*, *cd*)
    ¬in(empty, *cd*)
    in(addComponent(*cn*, *cm*, cd_1), *cd*) =
      in(*cm*, minus(*cd*, cd_1)) | in(*cn*, *cd*)
  implies converts [appearance: Contents, Coordinate → Illumination,
    in: Contents, Coordinate → Bool]
  exempts for all [*cd:* Coordinate] appearance(empty, *cd*)

Component: **trait**
  assumes Displayable **with** [View **for** T],
    Displayable **with** [Text **for** T],
    Displayable **with** [Figure **for** T]
  includes ComponentCoercion **with** [View **for** T],
    ComponentCoercion **with** [Text **for** T],
    ComponentCoercion **with** [Figure **for** T]

ComponentCoercion: **trait**
  assumes Displayable
  includes Displayable **with** [Component **for** T]
  introduces coerce: T → Component
  constrains Component **so that for all** [*t:* T, *cd:* Coordinate]
    appearance(coerce(*t*), *cd*) = appearance(*t*, *cd*)
    in(coerce(*t*), *cd*) = in(*t*, *cd*)

```
View: trait
    assumes Displayable with [ Picture for T ],
        Equality with [ PictureId for T ],
        Container with [ IdList for C, PictureId for E ],
        Coordinate
    includes Displayable with [ View for T ]
    introduces
        empty: → View
        addPicture: View, Coordinate, PictureId, Picture → View
        findPictures: View, Coordinate → IdList
        deletePicture: View, PictureId → View
    constrains View so that
        View generated by [ empty, addPicture ]
        for all [ v: View, cd, cd1: Coordinate, id, id1: PictureId, p: Picture ]
            appearance(addPicture(v, cd1, id, p), cd) =
                if in(p, minus(cd, cd1))
                    then appearance(p, minus(cd, cd1))
                    else appearance(v, cd)

            ¬in(empty, cd)
            in(addPicture(v, cd1, id, p), cd) = (in(p, minus(cd, cd1)) | in(v, cd))

            findPictures(empty, cd) = new
            findPictures(addPicture(v, cd1, id, p), cd) =
                if in(p, minus(cd, cd1))
                    then insert(id, findPictures(v, cd))
                    else findPictures(v, cd)

            deletePicture(empty, id) = empty
            deletePicture(addPicture(v, cd1, id1, p), id) =
                if id = id1 then v else addPicture(deletePicture(v, id), cd, id1, p)
    implies converts [ findPictures, deletePicture,
                appearance:View, Coordinate → Illumination,
                in:View, Coordinate → Bool ]
    exempts for all [ cd: Coordinate ] appearance(empty, cd)

Display: trait
    assumes Boundary, Transform, Illumination, Coordinate,
        Equality with [ PictureId for T ],
        Container with [ IdList for C, PictureId for E ]
    includes Picture, Contents, Component, View
```

Piece V

Writing Larch Interface Language Specifications

1. Introduction

Motivation

Current research in specifications is emphasizing the practical use of specifications in the programming process. People have already benefited from using informal specifications in most phases of this process. Writing informal specifications is widely accepted as a useful way of organizing ideas, documentating design decisions, and informally arguing the correctness of programs. Software design methods that include some form of informal specification have been in use in industry for some time [Caine and Gordon 75, Jackson 75, Katzan 76, Yourdon and Constantine 78].

Thus far, formal specifications have played a less influential role in the programming process than have informal specifications. We believe that using formal specifications early, in the design phase of the process, can be especially beneficial. A specification is formal if it is written in a language with explicitly and precisely defined syntax and semantics. Hence, one virtue of formal specifications is their precision. Precision leaves no room for ambiguity. The process of writing formal specifications can often reveal ambiguities in a client's problem statement and errors in a program's design. Uncovering bugs early can thus save the cost of uncovering them later during testing and debugging. Precision also implies that we can formally argue the correctness of programs. Another virtue of formal specifications is their amenability to machine-manipulation. With the help from appropriate machine-support, such as theorem provers, we can handle more specifications and more complex ones, and thus formally reason about a larger set of specifications and programs than if we had to rely on only pencil and paper.

In this piece we focus on the formal specifications of program modules. We are interested in specifying program modules as a means of specifying a program composed of them. Given a specification of a program module, a program designer can choose to use the module without knowing how it is to be implemented. Similarly, a programmer can implement the module without knowing how it is to be used. Thus, from either the designer's or implementer's point of view, replacing one correct implementation of the module by another should not affect the program's design.

Review: Larch's Two-Tiered Approach

The Larch languages were designed to support a two-tiered specification technique introduced in [Guttag and Horning 80] and elaborated in [Wing 83]. This approach separates the specification of underlying abstractions from the specification of state transformations. The specification of each program module has a component on each tier. The *Larch Shared Language* is used for the component that specifies underlying abstractions and a *Larch interface language* is used for the component that specifies state transformations.

We gain the following advantages by separating specifications into two tiers:

- A separation of concerns. Shared Language components can be written independently of interface language components. Application-oriented design decisions can be recorded separately from implementation-oriented decisions.
- Reuseability. Shared Language components can be reused by different interface language components. Some of them can be developed for particular applications; a few central ones can be useful in many applications.

The environment in which a program module is embedded, and hence the nature of its observable behavior, is likely to depend in fundamental ways on the semantic primitives of the programming language. Attempts to hide this dependence will make specifications more obscure to both the module's users and its implementers. Thus, we intentionally design each interface language to be suitable for a particular programming language, and keep the Shared Language independent of any programming language. To capitalize on this separation of a specification into two tiers, we isolate programming language dependent issues—such as side effects, error handling, and resource allocation—into the interface component of a specification.

We use the term "interface" because an interface specification defines only the observable behavior of a program module. Users of a module read its interface specification to understand its behavior, without considering its internal structure. We use the term "shared" because all the Larch interface languages rely on the same language to define underlying abstractions.

Focus of this Piece

This piece focuses on Larch interface language specifications. Its purpose is to explain in more detail what interface language specifications are, what they look like, and how they are intended to be written, used, and evaluated. A significant subgoal of this piece is to explain their interaction with Shared Language specifications. In Section 2 we present an informal description of an early version of Larch/CLU, an interface language for the programming language CLU [Liskov, *et al.* 77, Liskov, *et al.* 81]; in Section 3, we illustrate how to incrementally construct a two-tiered specification; in Section 4, we discuss some consequences of the two-tiered approach.

Related Work

Specification Methods: Formal specifications have been used extensively to describe simple programs and abstract data types, leading to two different approaches, sometimes referred to as "operational" and "definitional." A survey of these approaches can be found in [Liskov and Berzins 79].

In the operational approach, a specification gives a constructive definition of the program or abstract data type. Examples of the operational approach include Parnas's work on state-machines [Parnas 72], Robinson and Roubines's extensions to them with V-, O-, and OV-functions [Robinson and Roubine 77], Berzins's abstract models [Berzins 79], and Jones's model-oriented specifications [Jones 80].

In the definitional approach, the specification of a program or an abstract data type gives its required properties, rather than a method of constructing it. The definitional approach can be broken into two categories, sometimes referred to as "axiomatic" and "algebraic."

The axiomatic approach stems from Hoare's work on proofs of correctness of programs [Hoare 69] and of implementations of data types [Hoare 72], where predicate logic pre- and post-conditions are used for the specification of the input-output behavior of programs and of each operation of an abstract data type. Other work using the axiomatic approach is described in [Standish 73] and [Nakajima, *et al.* 80].

The algebraic approach uses axioms to specify properties of programs and abstract data types, but the axioms are restricted to equations. This approach defines data types to be heterogeneous algebras [Birkhoff and Lipson 70]. Much work has been done on the algebraic specification of abstract data types [ADJ 75, Guttag 75, Zilles, Burstall and Goguen 77, Ehrich 78, Wand 79, Kamin 83] including the handling of error values [Goguen 77, ADJ 75, Kapur 80], nondeterminism [Kapur 80], and parameterization [Thatcher, *et al.* 78, Goguen 81, Ehrig, *et al.* 80].

Our work is related to both these approaches. Interface languages are axiomatic and the Shared Language is algebraic.

Specification Languages: Some of the more widely-known specification languages are CLEAR [Burstall and Goguen 77, Burstall and Goguen 81], Iota [Nakajima, *et al.* 80], ACT-ONE [Ehrig and Mahr 85], SPECIAL [Robinson and Roubine 77], Z [Abrial 80], VDM's Meta-IV [Bjørner and Jones 78], Ina Jo [Scheid and Anderson 85], Gypsy [Good, *et al.* 78], and PAISLey [Zave 82]. Of these, the ones most closely related to ours are CLEAR, Iota, ACT-ONE, and SPECIAL.

CLEAR, Iota, and ACT-ONE support the definitional approach to describing abstract data types. Unlike the Larch Shared Language, CLEAR and Iota do not provide a simple way to specify side effects and error handling. CLEAR and ACT-ONE are based on initial algebra semantics; the Larch Shared Language is not. CLEAR, Iota, and ACT-ONE do not isolate the programming-language-dependent parts of a specification.

SPECIAL is based on the operational approach, but is closely related to our two-tiered viewpoint. It separates an "assertion" part, analogous to our Shared Language component, from a "specification" part, analogous to our interface language component. However, in SPECIAL a type is restricted to be either a primitive type, a subtype, or a structured type, each of which comes with a set of pre-defined functions. Larch does not restrict the assertion language to a fixed set of primitives, and allows the specifier to use the Shared Language to define exactly the desired operators. Since the assertion language in SPECIAL is restricted, most of the work of writing a specification is done in the specification part. We take the opposite viewpoint and expect most of the work of writing a specification to be done in the Shared Language component.

2. An Informal Look at a Larch/CLU Interface Language

This section presents part of an interface language for the programming language CLU. Instead of giving a formal description of Larch/CLU,* we will illustrate its salient features through some simple examples. Its complete formal definition can be found in [Wing 83].

The meaning of a Larch interface language is dependent on both the Larch Shared Language and a programming language. Pieces I–IV have presented the Shared Language. The next section reviews those parts of CLU that are needed to understand our example interface specifications.† We refer the reader to [Liskov, *et al.* 81] for details about CLU. Then we give examples of both a Larch/CLU procedure specification and a Larch/CLU cluster specification.

An Overview of CLU

CLU has the primitive notions of *object* and *state*. An object is an entity that can be manipulated by a program. Two important properties of an object are its *type*, which never changes, and its *value*, which may change. A state consists of a set of objects, a mapping from program variables (object identifiers) to objects, and a mapping from objects to values. Two important observable state changes are when a new object is created and when the value of an existing object changes. An object whose value can change is said to be *mutable*; one whose value cannot change is said to be *immutable*. A type is mutable if objects of that type are mutable. For example, integers are immutable, but arrays are mutable in CLU.

In CLU, an object, A, can be the value of another object, B, in which case we say "B contains A." Sharing of objects arises when two or more objects contain the same object. Because of sharing of mutable objects, it is not sufficient that the value of a containing object refer to the value of the contained object; it must refer to the identity of the contained object itself. Therefore, we must be able to distinguish in our specifications between an object's identity and its value.

It is important not to confuse an object and its type, which are CLU concepts, with a term and its sort, which are Larch Shared Language concepts. The connection between the CLU and the Larch Shared Language concepts is that (typed) objects have values that are denotable by (sorted) terms. Through the Larch/CLU interface specifications of

* The Larch/CLU used in this Piece is a predecessor of the one in Piece I. Although the surface syntax is somewhat different, the underlying semantics is essentially the same.

† In this piece we ignore the following features of CLU: iterators, own data, and parameterized modules. They are all carefully treated in [Wing 83].

choose = **proc** (s: set) **returns** (i: int)
  uses SetOfE
    requires $\neg$isEmpty(s_{pre})
    modifies at most [s]
    ensures has(s_{pre}, i_{post}) & s_{post} = remove(s_{pre}, i_{post})
  end

Figure 1. A Procedure Specification

procedures and clusters, we establish a link between the values that objects can have and the terms defined by Shared Language components.

A CLU program consists of a set of modules, each of which is either a *procedure* or a *cluster*. A procedure performs an action on a set of objects, and terminates returning a set of objects. Communication between a procedure and its invoker occurs through these objects. A cluster names a type and defines a set of procedures that create and manipulate objects of that type. Users of this type are constrained to treat objects of the type abstractly. That is, objects can be manipulated only via the procedures defined by the cluster so, in particular, information about how objects are represented may not be used.

Larch/CLU Procedure Specifications

Figure 1 gives a Larch/CLU specification of a choose procedure that selects a member of a set, removes it, and returns it. It consists of a *header*, a *link* to its Shared Language component, and a *body*. The header indicates that the input argument is of type set, and the output argument is of type int. The identifiers, s and i, denote objects, not values. The link from the interface component to the shared component is given by the *used trait*, SetOfE, which is presented in Figure 2. The body contains a pre/mutates/post triple. The pre-condition of choose is an assertion that is satisfied if the initial value of the input argument is not empty. The **modifies at most** clause asserts that the choose procedure may mutate no object other than the object bound to s. The post-condition is an assertion about the initial and final values of the set object and the final value of the int object. The operator names, isEmpty, has, and remove, and the meaning of the equality symbol, =, all come from SetOfE.

Associated with a procedure specification is the predicate,

$$\text{PRE} \Rightarrow (\text{MUTATES } \& \text{ POST})$$

where PRE and POST are the assertions in the **requires** and **ensures** clauses, respectively, and MUTATES is the assertion associated with the **modifies at most** clause. The clause

```
SetOfE: trait
    includes Integer
    introduces
        empty: → SI
        add: SI, E → SI
        remove: SI, E → SI
        has: SI, E → Bool
        isEmpty: SI → Bool
        card: SI → Int
    constrains empty, add, remove, has, isEmpty, card so that
        SI generated by [empty, add]
        for all [s: SI, e, e1: E]
            remove(empty, e) = empty
            remove(add(s, e), e1) =
                if e = e1 then remove(s, e1) else add(remove(s, e1), e)
            ¬has(empty, e)
            has(add(s, e), e1) = if e = e1 then true else has(s, e1)
            isEmpty(empty)
            ¬isEmpty(add(s, e))
            card(empty) = 0
            card(add(s, e)) = if has(s, e) then card(s) else 1 + card(s)
```

Figure 2. SetOfE Trait

modifies at most [$x_1, \ldots, x_n$] asserts that the procedure changes the value of no object in the environment of the caller except possibly some subset of $\{x_1, \ldots, x_n\}$.

The following points are important to notice about a procedure specification:

- We distinguish between an object and its value by using a plain object identifier to denote an object, and a subscripted object identifier to denote its value in a state.
- We distinguish between the initial and final values of an object by using an object identifier subscripted by *pre* to denote the object's initial value, and subscripted by *post* to denote its final value. Thus the assertion $s_{pre} = s_{post}$ says that the value of the object s is unchanged.
- The headers for a CLU procedure and a CLU procedure specification are intentionally similar. The only difference is that object identifiers, such as i, are introduced for returned objects in the header of a procedure specification. This is to provide a way to denote them in the assertions.

set = **cluster is** pair, union, intersect, member, size
    uses SetOfE
    provides mutable set **from** SI

    pair = **proc** (i, j: int) **returns** (s: set)
        ensures (s_{post} = add(add(empty, i_{pre}), j_{pre})) & **new** [s]
    end

    union = **proc** (s1, s2: set)
        modifies at most [s2]
        ensures ∀ j: E [has($s2_{post}$, j) = (has($s1_{pre}$, j) | has($s2_{pre}$, j))]
    end

    intersect = **proc** (s1, s2: set)
        modifies at most [s2]
        ensures ∀ j: E [has($s2_{post}$, j) = (has($s1_{pre}$, j) & has($s2_{pre}$, j))]
    end

    member = **proc** (s: set, i: int) **returns** (b: bool)
        ensures b_{post} = has(s_{pre}, i_{pre})
    end

    size = **proc** (s: set) **returns** (i: int)
        ensures i_{post} = card(s_{pre})
    end

end set

Figure 3. A Set Cluster Specification

- The name of the used trait denotes the Shared Language component.
- The **modifies at most** clause is an assertion that is given meaning as if it were conjoined to the post-condition (see above). It is syntactically separated from the post-condition to highlight a procedure's potential side effect on the values of objects. It is an example of a *special assertion*; each interface language comes equipped with its own set of special assertions. They can be regarded as syntactic sugar for first-order assertions about state.

A Larch/CLU Cluster Specification

Figure 3 gives a Larch/CLU specification for a *set* cluster. It consists of a header, a link to its Shared Language component, and a body. The header consists of the type identifier, set, and a list of the procedure identifiers, pair, union, intersect, member, and size.

Notice that set is the name of a type, not a sort. It is also the name of the cluster specification and is different from any trait name. The link from the interface component to the shared component is given by the used trait, SetOfE, and a **provides** clause. SetOfE supplies all sort and operator identifiers that appear in the assertions of the procedure specifications of the cluster specification. For example, the sort identifier, E, which appears in the post-condition of union, comes from SetOfE, and is used for terms denoting integer values. The **provides** clause gives a mapping from the type identifier, set, to the sort identifier, SI, which also comes from SetOfE. This type-to-sort mapping determines the values over which set objects can range. All set objects are restricted to values denotable by terms of sort SI. The **provides** clause also indicates whether the type is mutable or not. The body of a cluster specification consists of specifications of the procedures, which are of the form described for procedure specifications.

Two additional features of Larch/CLU are illustrated in the specification of pair: omitted **modifies at most** clauses, and **new** assertions. First, the omission of a **modifies at most** clause means that no objects may be mutated by the procedure—for each call, the value of each object must be the same on return as on entry. Second, we use **new** assertions to indicate objects that must not be the same as any existing object. For example, pair's specification states that it must not return a set object that existed when pair was invoked.

Let us consider writing a different set cluster specification, set2, that defines a different set type—one with a slightly different specification for the intersect procedure. Let the specification of set2 be the same as that of set in Figure 3 except that intersect2 returns the intersection of its two arguments only if they are not disjoint; otherwise, it terminates exceptionally, signaling "disjoint." That is, let intersect2 be:

intersect2 = **proc** (s1, s2: set) **signals** (disjoint) — drop
    requires ~~$\neg\exists$ j: E [has($s1_{pre}$, j) & has($s2_{pre}$, j)]~~
    modifies at most [s2]
    ensures
        normally $\forall$ j: E [has($s2_{post}$, j) = (has($s1_{pre}$, j) & has($s2_{pre}$, j))]**except**
        signals disjoint **when** $\neg\exists$ j: E [has($s1_{pre}$, j) & has($s2_{pre}$, j)]
            ensuring modifies nothing

end

Even though set and set2 specify different types, they both use the same trait, SetOfE. Therefore, set objects defined by set of Figure 3 range over values denoted by the same terms as set objects defined by set2. This difference illustrates that there is a clear distinction between a sort identifier and a type identifier. Although the trait SetOfE introduces the term empty of sort SI to denote the "empty" value, no object of type set2 will ever have such a value since only nonempty set objects can be constructed by set2's (constructor) operations, pair, union, and intersect2.

An additional feature of Larch/CLU is illustrated by the specification of intersect2. CLU procedures may either terminate normally or terminate by signalling an exception. The clause beginning with **normally** asserts that if s1 and s2 have no element in common, intersect2 raises the exception disjoint and modifies nothing. Otherwise, intersect2 returns normally and modifies s2 so that its final value is the intersection of the initial values of s1 and s2. Demarcating these individual cases enhances the readability of the specification and disciplines the specifier to consider all possible cases in a stylized way.

3. Incrementally Writing an Interface Specification

As mentioned in the Introduction, writing Larch specifications is intended to occur during the design process with the help of machine-support. In this section, we will illustrate how to write an interface specification following Larch's two-tiered approach as intertwined with a typical top-down design process. We will also mention some of the machine-support a specifier might expect as a two-tiered specification is written.

Following the Approach

We sketch below a typical top-down design strategy that could be used in following the two-tiered approach.

- Develop an approximate intuition of the problem to be solved. This requires close, often verbal, interaction with the client who is posing the problem.
- Decide on the major abstractions.
  - Interface language tier: Write the header information of the interface language components.
  - Shared Language tier: Write the syntactic information of the Shared Language components of the specification, namely, the sort identifiers, operator identifiers, and operator signatures.
- Fill in the blanks.
  - Interface language tier: Fill in the information in the bodies of the interface language components, by writing the assertions for the bodies of the procedure specifications. Note any additional operator and sort identifiers used, so they can be defined in the Shared Language components.
  - Link between the two tiers: Define the explicit link to the Shared Language components of the specification.
  - Shared Language tier: Fill in the semantic information in the bodies of the Shared Language components of the specification, namely, the theory of equality for terms.
- Check one's understanding of the problem and its formalization; repeat previous steps until they converge.

During this process of writing a specification, the specifier should also evaluate it for certain properties, such as consistency and completeness. Checking for these properties as a specification develops can increase confidence that a specification is on the right

Interface Language Components

```
dictionary = cluster is ...
    uses DictVals
    provides dictionary from ...
            ...
    end dictionary

word = cluster is ...
    uses WordVals
    provides word from ...
            ...
    end word

definition = cluster is ...
    uses DefVals
    provides definition from ...
            ...
    end definition
```

Shared Language Components

```
DictVals: trait
    introduces
            ...
    constrains
            ...

WordVals: trait
    introduces
            ...
    constrains
            ...

DefVals: trait
    introduces
            ...
    constrains
            ...
```

Figure 4. Dictionary Specification: Snapshot 1

track. In the example of the next subsection, we will describe a check for one such property, *totality*, to illustrate how feedback from evaluating a specification can influence the specifier. In Section 4, we discuss two other checkable properties of interfaces, *protection* and *nondeterminism*.*

As with any overall design method, many iterations over the steps may be necessary. Writing a specification sharpens a specifier's intuition of the problem. Hidden design decisions surface. Addressing postponed decisions often requires modifications of decisions made earlier. Specifiers should be willing to discard large chunks of a specification in the process of refining the abstractions. Specifiers (as well as designers and programmers) are often reluctant to start anew or to try alternative tactics. However, good support from the machine should help to overcome this reluctance.

An Example Illustrating the Two-Tiered Approach

In this section we trace one iteration of the strategy outlined in the previous section with a series of snapshots that show the incremental development of a specification. We use a simple example to keep the details from obscuring the points we wish to make.

Suppose we want to write a specification of a dictionary that contains the definitions of words and that can be used to check the spelling of words. For simplicity, let us assume that a word can appear only once in a dictionary, and each word has exactly one definition. Furthermore, if a word is not in the dictionary, then the word is either misspelled or unknown to the dictionary (e.g., a rarely used word might not be found in an abridged dictionary). Intuitively, a dictionary is like a table that stores key-value pairs, where words are the keys and definitions are the values.

From this informal description of a dictionary and an intuitive understanding of its usage, we next have to decide on the major abstractions. We choose the data types of interest to be dictionary, word, and definition. Therefore, we need to write cluster specifications for each of the three types and appropriate traits for the values of objects of each type. Since we need a used trait for each cluster specification, let us name them DictVals, WordVals, and DefVals. Figure 4 depicts the situation so far. We are presuming the use of a syntax-directed specification editor that displays the templates shown in the figure and prompts us to fill in each "...".

We begin by further developing the dictionary cluster specification and the corresponding DictVals trait, and postpone developing the other specification components until later. Given the informal description of the usage of a dictionary, we have to decide what

---

* A more detailed discussion of these and other properties of interface specifications can be found in [Wing 83, Wing 84].

```
dictionary = cluster is create, add_word, get_definition, check_word
        uses DictVals
        provides dictionary from ...
            create = proc () returns (d: dictionary)
                    requires ...
                    modifies at most ...
                    ensures ...
            end
            add_word = proc (d: dictionary, w: word, def: definition)
                    requires ...
                    modifies at most ...
                    ensures ...
            end
            get_definition = proc (d: dictionary, w: word) returns (def: definition)
                    requires ...
                    modifies at most ...
                    ensures ...
            end
            check_word = proc (d: dictionary, w: word) returns (b: bool)
                    requires ...
                    modifies at most ...
                    ensures ...
            end
        end dictionary
DictVals: trait
        introduces
            ...
        constrains
            ...
```

Figure 5. Dictionary Specification: Snapshot 2

operations would most likely be performed on dictionaries. Some of the table-like operations we might want to perform are to create a dictionary, add a new word and its definition to a dictionary, get the definition of a word, and check to see if a word is in a dictionary. After filling in some syntactic information for dictionary, we have the situation as shown in Figure 5. Visible changes from one snapshot to the next are shown in italics.

Next we start filling in the bodies of the procedure specifications and simultaneously generate sort and operator identifiers that must be supplied by DictVals. We start with create. We do not want any restrictions on the computation state in creating a new dictionary, nor do we want any objects to be mutated; we want the value of the returned dictionary to be empty and we want the dictionary itself to be some new object. So for create we have (notice the deletion of the **modifies at most** clause):

> create = **proc** () **returns** (d: dictionary)
>   **ensures** (d_{post} = empty) & **new** [d]
> **end**

In order to denote the empty value of a dictionary, we used the operator identifier, empty, in create's post-condition. The empty operator must be defined in DictVals by first giving empty a signature, which in turn causes us to introduce a sort identifier, D, to which the type identifier dictionary can map. Consequently, we can define the type-to-sort mapping in the **provides** clause of dictionary. We now have the situation shown in Figure 6.

Next we turn to filling in the body of add_word. We want to add a word and its definition to a dictionary only if the word is not already in the dictionary. We state this constraint in the pre-condition of add_word. We have:

> add_word = **proc** (d: dictionary, w: word, def: definition)
>   **requires** ¬isIn(d_{pre}, w_{pre})
>   **modifies at most** [d]
>   **ensures** d_{post} = insert(d_{pre}, w_{pre}, def_{pre})
> **end**

Notice a design decision we have made: by allowing the dictionary input to add_word to be possibly mutated, we have decided to make dictionary a mutable type. We document this decision in the **provides** clause of dictionary with the keyword **modifies at most**.

The definitions of the functions isIn and insert are still pending in DictVals. To give a signature for insert, we introduce the sort identifiers W and Dfn, corresponding to the types word and definition map, respectively. Thus, we can refine the specifications of the types word and definition in Figure 4 by completing their **provides** clauses. We can also write equations in DictVals to define the operators already introduced. Figure 7 shows the situation so far for dictionary and DictVals.

```
dictionary = cluster is create, add_word, get_definition, check_word
    uses DictVals
    provides dictionary from D
        create = proc () returns (d: dictionary)
                ensures (d_post = empty) & new [ d ]
        end
        add_word = proc (d: dictionary, w: word, def: definition)
                requires ...
                modifies at most ...
                ensures ...
        end
        get_definition = proc (d: dictionary, w: word) returns (def: definition)
                requires ...
                modifies at most ...
                ensures ...
        end
        check_word = proc (d: dictionary, w: word) returns (b: bool)
                requires ...
                modifies at most ...
                ensures ...
        end
    end dictionary

DictVals: trait
        introduces
            empty: → D
            ...
        constrains
            ...
```

Figure 6. Dictionary Specification: Snapshot 3

```
dictionary = cluster is create, add_word, get_definition, check_word
    uses DictVals
    provides mutable dictionary from D

        create = proc () returns (d: dictionary)
                ensures (d_post = empty) & new [ d ]
        end

        add_word = proc (d: dictionary, w: word, def: definition)
                requires ¬isIn(d_pre, w_pre)
                modifies at most [ d ]
                ensures d_post = insert(d_pre, w_pre, def_pre)
        end

        get_definition = proc (d: dictionary, w: word) returns (def: definition)
                requires ...
                modifies at most ...
                ensures ...
        end

        check_word = proc (d: dictionary, w: word) returns (b: bool)
                requires ...
                modifies at most ...
                ensures ...
        end

    end dictionary

DictVals: trait
    introduces
        empty: → D
        insert: D, W, Dfn → D
        isIn: D, W → Bool
        ...
    constrains empty, insert, isIn so that
        for all [d: D, w, w1: W, dfn: Dfn]
        ¬isIn(empty)
        isIn(insert(d, w, dfn), w1) = (w = w1) | isIn(d, w1)
        ...
```

Figure 7. Dictionary Specification: Snapshot 4

```
dictionary = cluster is create, add_word, get_definition, check_word
    uses DictVals
    provides mutable dictionary from D

        create = proc () returns (d: dictionary)
                ensures (d_post = empty) & new [ d ]
        end

        add_word = proc (d: dictionary, w: word, def: definition)
                requires ¬isIn(d_pre, w_pre)
                modifies at most [ d ]
                ensures d_post = insert(d_pre, w_pre, def_pre)
        end

        get_definition = proc (d: dictionary, w: word) returns (def: definition)
                requires isIn(d_pre, w_pre)
                ensures def_post = lookup(d_pre, w_pre)
        end

        check_word = proc (d: dictionary, w: word) returns (b: bool)
                ensures b_post = isIn(d_pre, w_pre)
        end

    end dictionary

DictVals: trait
        introduces
            empty: → D
            insert: D, W, Dfn → D
            isIn: D, W → Bool
            lookup: D, W → Dfn
        constrains empty, insert, isIn, lookup so that
            for all [d: D, w, w1: W, dfn: Dfn]
            ¬isIn(empty)
            isIn(insert(d, w, dfn), w1) = (w = w1) | isIn(d, w1)
            lookup(insert(d, w, dfn), w1) = if w = w1 then dfn else lookup(d, w1)
```

Figure 8. Dictionary Specification: Snapshot 5

Continuing this process by filling in the bodies of get_definition and check_word causes us to introduce only one more function identifier, lookup. After adding an equation to define lookup in DictVals, we end up with a dictionary specification and a DictVals trait as shown in Figure 8.

Evaluating the Dictionary Cluster Specification So Far: At this point, before proceeding to the word and definition cluster specifications, it is worth reflecting on the dictionary specification we have just written. During the incremental development of a specification, it is useful to see if it can be improved and to check whether we are still on the right track. In this section we will discuss the evaluation of interface specifications for the property of *totality*.

Notice that the pre-condition of the add_word specification is not (identically) true, which means that the behavior of an add_word procedure is left unspecified for some possible states in which it can be invoked. We say the add_word specification is not *total* [Wing 83]. Upon checking add_word for totality, we may be inclined to make it total and handle the case for which the word we attempt to add to the dictionary is already in the dictionary. We might modify add_word to terminate exceptionally in this case:

add_word = **proc** (d: dictionary, w: word, def: definition) *signals (alreadyIn)*
  modifies at most [d]
  ensures
    normally d_{post} = insert(d_{pre}, w_{pre}, def_{pre}) **except**
    signals alreadyIn when isIn(d_{pre}, w_{pre}) ensuring modifies nothing
end

Similarly, get_definition is also not total. We choose to make it total and handle the case in which a word is not in the dictionary:

get_definition = **proc** (d: dictionary, w: word) **returns** (def: definition)
    signals (wordNotIn)
  ensures
    normally def_{post} = lookup(d_{pre}, w_{pre}) *except*
    signals wordNotIn when $\neg$*isIn(d_{pre}, w_{pre})*
end

If we were to decide to leave a procedure specification not total, then the implementer would be free to choose the behavior of the procedure for the unspecified cases. Unfortunately, implementers may often forget to handle unspecified cases, which may lead to surprising or erroneous behavior. On the other hand, it may not be necessary to handle unspecified cases that can never arise. For example, the choose procedure specification of Figure 1 is not total. If it were defined to operate on sets as defined by the set2 cluster specification described in Section 2, there would be no need to handle the empty set since it would never arise (assuming a correct implementation of set2).

Completing the Remaining Interface Specifications: We now turn to filling in the blanks for the word and definition cluster specifications and the WordVals and DefVals traits. Recall that the informal description of the usage of a dictionary requires that we must be able to check the spelling of a given word against the spellings of the words in the dictionary. This requirement implies that the word cluster must have a procedure that tests for equality between two words. No other requirements or constraints were made on words, such as if words are sequences of only alphabetic characters (perhaps numerals and punctuation symbols are allowed) or if there exists a "null" word. Therefore, until further constraints are made by the client, it suffices to include in the word cluster specification a specification of an equal procedure.

Finally, we turn to definition and DefVals. We have even less information about definitions of words in a dictionary than we have about words. For instance, we do not know whether definitions are sentences, phrases, or combinations of both, or whether they must conform to a fixed format. The only information we can include in the definition cluster specification is the type-to-sort mapping in the **provides** clause. Recall that we generated this information when we introduced the insert function for the dictionary cluster specification.

We have essentially gone through one iteration of the strategy as outlined above. At this point, we need to return to the client and ask for more information. After further elaboration of the problem description, appropriate additions and modifications can then be made to the specification.

4. Implications of the Two-Tiered Approach

Interactions Between the Two Tiers

Interface specifications describe what is to be implemented; traits do not. Operations defined in interface specifications are intended to be implemented by procedures, but operators of traits are not. Thus, for example, the pair operation of the set type as specified in Figure 3 is to be implemented by some CLU procedure, but the add operator of the SetOfE trait is not.

When suitable abstractions have been defined in the Shared Language components, the interface language components of specifications often appear to be trivial. In order to keep the interface language component simple, we generally place the complexity of a two-tiered specification in the Shared Language component. Complexity in the interface component may be a symptom that an abstraction is missing in the shared component. For example, it might have been better to define set intersection in trait SetOfE (Figure 2), rather than in intersect's **ensures** clause (Figure 3).

Protection and nondeterminism both illustrate ways in which the two tiers interact. Protection is related to the sufficient-completeness of an algebraic specification [Guttag 75]. The Larch Shared Language does not require that traits be sufficiently-complete, and provides a construct, **exempts**, for indicating that the meaning of certain terms need not be defined. We avoid using such terms in interface language components by supplying pre-conditions to ensure that the meaning of an interface does not depend on the meaning of exempt terms. For example, the DictVals trait is not sufficiently-complete because the meaning of lookup(empty, w) is left unconstrained. However, a **requires** clause ensures that get_definition's meaning is independent of the meaning of lookup(empty, w). Thus, get_definition is *protective* of DictVals.

Nondeterminism deals with a different kind of incompleteness—that of underconstraining final values of objects. For example, the specification of choose in Figure 1 is nondeterministic. Nondeterminism cannot be introduced by traits. The mathematical basis of algebra and of the Larch Shared Language depends on the ability to freely substitute equals for equals. This property would be destroyed if trait operators were allowed to represent "nondeterministic functions."

Important Properties for the Two-Tiered Approach

Most of the advantages of two-tiered specifications are independent of the details of Larch. The properties that make the Larch family of languages well-suited to the two-tiered approach are as follows:

- There is a clear syntactic and semantic distinction between specifications of properties of underlying abstractions and specifications of properties of program components.
- The set of abstractions used in specifying interfaces is open-ended, yet each abstraction is precisely defined.
- Specifications of abstractions can be easily reused, even for program components written in different languages.
- Each interface language can be optimized for communicating the important properties of interfaces in a particular programming language.
- The most delicate piece of the specification language design can be shared by specification languages for many different programming languages.

Postlude

Acknowledgments

Butler Lampson, Mary-Claire van Leunen, and Søren Prehn have been diligent in helping us to improve the exposition of our ideas. Several members of the Larch Project have contributed ideas, criticism, and trial implementations to the development of Larch. Dave Detlefs, Randy Forgaard, Ron Kownacki, and Joe Zachary deserve special thanks.

IFIP Working Group 2.3 (Programming Methodology) provided both a continuing education and a constructively critical sounding board. Jean-Raymond Abrial, Sharon Anderson, Rod Burstall, Susan Gerhart, Cliff Jones, Barbara Liskov, Bill McKeeman, Dave Musser, Doug Ross, Mary Shaw, Jim Thatcher, and Steve Zilles were all, in their own ways, significant influences on this work.

References

[Abrial 80] J. Abrial
The Specification Language Z: Syntax and Semantics
Oxford University Computing Laboratory, Programming Research Group, Apr. 1980.

[ADJ 75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright
"Abstract Data Types as Initial Algebras and Correctness of Data Representations"
Proc. ACM Conf. Computer Graphics, Pattern Recognition and Data Structures
May 1975, 89–93.

[ADJ 78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner
"Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in R. T. Yeh (ed.)
Current Trends in Programming Methodology, Vol. IV, Data Structuring
Prentice-Hall, 1978.

[Atreya 82] S. K. Atreya
"Formal Specification of a Specification Library"
S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1982.

[Berzins 79] V. A. Berzins
"Abstract Model Specifications for Data Abstractions"
Laboratory for Computer Science, Massachusetts Institute of Technology, MIT/LCS/TR-221, July 1979.

[Birkhoff and Lipson 70] G. Birkhoff and J. D. Lipson
"Heterogeneous Algebras" *J. Combinatorial Theory*, vol. 8, 1970, 115–133.

[Bjørner and Jones 78] D. Bjørner and C. G. Jones
The Vienna Development Method: the Meta-language
Springer-Verlag Lecture Notes in Computer Science, vol. 61, 1978.

[Burstall and Darlington 77] R. M. Burstall and J. Darlington
"A Transformation System for Developing Recursive Programs"
J. ACM, vol. 24, no. 1, Jan. 1977, 44–67.

[Burstall and Goguen 77] R. M. Burstall and J. A. Goguen
"Putting Theories Together to Make Specifications"
Proc. 5th International Joint Conference on Artificial Intelligence
1977, 1045–1058.

[Burstall and Goguen 81] R. M. Burstall and J. A. Goguen
"An Informal Introduction to Specifications Using CLEAR"
in R. Boyer and J. Moore (eds.), *The Correctness Problem in Computer Science*
Academic Press, New York, 1981, 185–213.

[Caine and Gordon 75] S. H. Caine and E. K. Gordon
"PDL—A Tool for Software Design"
Proc. 1975 NCC, 271–276.

[Ehrich 78] H.-D. Ehrich
"Extensions and Implementations of Abstract Data Type Specifications"
Proc. Mathematical Foundations of Computer Science 1978
Springer-Verlag Lecture Notes in Computer Science, vol. 64, 155–164.

[Ehrig and Kreowski 82] H. Ehrig and H.-J. Kreowski
"Parameter Passing Commutes with Implementation of Parameterized Data Types"
Proc. 9th Colloquium on Automata, Languages and Programming
Springer Verlag, 1982, 197–211.

[Ehrig and Mahr 85] H. Ehrig and B. Mahr
Fundamentals of Algebraic Specification 1: Equations and Initial Semantics
Springer-Verlag, EATCS Monographs on Theoretical Computer Science, vol. 6, 1985.

[Ehrig, *et al.* 80] H. Ehrig, H.-J. Kreowski, J. Thatcher, E. Wagner, and J. Wright
"Parameterized Data Types in Algebraic Specification Languages"
Automata, Languages, and Programming
Springer-Verlag Lecture Notes in Computer Science, vol. 85, July 1980, 157–168.

[Forgaard 85] R. Forgaard
"A Program for Generating and Analyzing Term Rewriting Systems"
S.M. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, MIT/LCS/TR-343, 1985.

[Goguen 77] J. A. Goguen
"Abstract Errors for Abstract Data Types"
Proc. IFIP Working Conference on Formal Basis of Programming Concepts
North-Holland, 1977, 21.1–21.32.

[Goguen and Parsaye-Ghomi 81] J. A. Goguen and K. Parsay-Ghomi
"Algebraic Denotational Semantics Using Parameterized Abstract Modules"
Technical Report CSL-119, Stanford Research Institute, Feb. 1981.

[Good, *et al.* 78] D. I. Good, R. M. Cohen, C. G. Hoch, L. W. Hunter, and D. F. Hare
"Report on the Language Gypsy, Version 2.0"
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, University of Texas at Austin, Sept. 1978.

[Guttag 75] J. V. Guttag
"The Specification and Application to Programming of Abstract Data Types"
Ph.D. Thesis, Computer Science Department, University of Toronto, 1975.

[Guttag and Horning 78] J. V. Guttag and J. J. Horning
"The Algebraic Specification of Abstract Data Types"
Acta Informatica, vol. 10, 1978, 27–52.

[Guttag and Horning 80] J. V. Guttag and J. J. Horning
"Formal Specification as a Design Tool"
Proc. 7th ACM Symposium on Principles of Programming Languages
Jan. 1980, 251–261.

[Guttag and Horning 83a] J. V. Guttag and J. J. Horning
"An Introduction to the Larch Shared Language"
Proc. IFIP Congress '83, North-Holland, 1983.

[Guttag and Horning 83b] J. V. Guttag and J. J. Horning
Preliminary Report on the Larch Shared Language
Technical Report MIT/LCS/TR-307, Laboratory for Computer Science, Massachusetts Institute of Technology, October 1983;
also issued as Technical Report CSL-83-6, Computer Science Laboratory, Xerox Palo Alto Research Center, December 1983.

[Guttag and Horning 85a] J. V. Guttag and J. J. Horning
"Report on the Larch Shared Language"
Science of Computer Programming, to appear.

[Guttag and Horning 85b] J. V. Guttag and J. J. Horning
"A Larch Shared Language Handbook"
Science of Computer Programming, to appear.

[Guttag, Horning, and Wing 82]
J. V. Guttag, J. J. Horning, and J. M. Wing
"Some Notes on Putting Formal Specifications to Productive Use"
Science of Computer Programming, vol. 2, Dec. 1982, 53–68.

[Guttag, Horning, and Wing 85]
John V. Guttag, James J. Horning, and Jeannette M. Wing,
"The Larch Family of Specification Languages"
IEEE Software, vol. 2., no. 4, Sept. 1985.

[Guttag and Liskov 86] J. V. Guttag and B. H. Liskov
Abstraction and Specification in Program Design
MIT Press/McGraw Hill, to appear 1986.

[Hehner 84] E. C. R. Hehner
"Predicative Programming, Parts I and II"
Comm. ACM, vol. 27, Feb. 1984, 134–151.

[Hoare 69] C. A. R. Hoare
"An Axiomatic Basis for Computer Programming"
Comm. ACM, vol. 12, no. 10, Oct. 1969, 576–583.

[Hoare 72] C. A. R. Hoare
"Proof of Correctness of Data Representations"
Acta Informatica, vol. 1, 1972, 271–281.

[Horning 85] J. J. Horning
"Combining Algebraic and Predicative Specifications in Larch"
in [TAPSOFT 85] vol. 2, 12–26.

[Jackson 75] M. A. Jackson
Principles of Program Design. Academic Press, 1975.

[Jones 77] C. B. Jones
"Implementation Bias in Constructive Specifications," manuscript, Sept. 1977.

[Jones 80] C. B. Jones
Software Development: A Rigorous Approach. Prentice-Hall International, 1980.

[Kamin 83] S. Kamin
"Final Data Types and Their Specification"
ACM Trans. Programming Languages and Systems, vol. 5, no. 1, Jan. 1983, 97–121.

[Kapur 80] D. Kapur
"Towards a Theory for Abstract Data Types"
Laboratory for Computer Science, Massachusetts Institute of Technology,
Technical Report MIT/LCS/TR-237, May 1980.

[Katzan 76] H. Katzan, Jr.
Systems Design and Documentation: An Introduction to the HIPO Method
Van Nostrand Reinhold, 1976.

[Kownacki 84] R. Kownacki
"Semantic Checking of Formal Specifications"
S.M. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1984.

[Lamport 85] Leslie Lamport
"What It Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority"
Proc. 12th ACM Symposium on the Principles of Programming Languages, Jan. 1985.

[Lescanne 83] P. Lescanne
"Computer Experiments with the REVE Term Rewriting System Generator"
Proc. 10th ACM Symposium on the Principles of Programming Languages
Jan. 1983, 99–108.

[Liskov and Berzins 79] B. H. Liskov and V. Berzins
"An Appraisal of Program Specifications"
in P. Wegner (ed.), *Research Directions in Software Technology*
MIT Press, 1979, 276–301.

[Liskov, *et al.* 77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert
"Abstraction Mechanisms in CLU"
Comm. ACM, vol. 20, no. 8, Aug. 1977, 564–576.

[Liskov, *et al.* 81]
B. Liskov, R. Atkinson, T. Blum, E. Moss, C. Schaffert, R. Scheifler, A. Snyder
CLU Reference Manual
Springer-Verlag Lecture Notes in Computer Science, vol. 114, 1981.

[Luckham and von Henke 85] David Luckham and Friedrich W. von Henke
"An Overview of Anna, a Specification Language for Ada"
IEEE Software, vol. 2, no. 2, Mar. 1985, 9–22.

[Meyer 85] Bertrand Meyer
"On Formalism in Specifications"
IEEE Software, vol. 2, no. 1, Jan. 1985, 6–26.

[Musser 80] D. R. Musser
"Abstract Data Type Specification in the Affirm System"
IEEE Trans. Software Engineering, vol. SE-6, no. 1, Jan. 1980, 24–32.

[Nakajima, *et al.* 80] R. Nakajima, M. Honda, and H. Nakahara
"Hierarchical Program Specification and Verification—A Many-sorted Logical Approach"
Acta Informatica, vol. 14, 1980, 135–155.

[Parnas 72] D. L. Parnas
"A Technique for Software Module Specification with Examples"
Comm. ACM, vol. 15, no. 5, May 1972, 330–336.

[Robinson and Roubine 77] L. Robinson and O. Roubine
"SPECIAL—A Specification and Assertion Language"
Technical Report CSL-46, Stanford Research Institute, 1977.

[Sannella and Tarlecki 85] Donald Sannella and Andrzej Tarlecki
"Some Thoughts on Algebraic Specification," to appear.

[Scheid and Anderson 85] J. Scheid and S. Anderson
"The Ina Jo Specification Language Reference Manual"
Technical Report TM-(L)-6021/001/01, System Development Corporation, March 1985.

[Shaw 84] Mary Shaw
"Abstraction Techniques in Modern Programming Languages"
IEEE Software, vol. 1, no. 4, October 1984, 10–26.

[Standish 73] T. A. Standish
"Data Structures: An Axiomatic Approach"
Technical Report 2639, Bolt Beranek, and Newman, Inc., Aug. 1973.

[TAPSOFT 85]
Proc. International Joint Conference on Theory and Practice of Software Development
Volume 1: *Mathematical Foundations of Software Development*
Volume 2: *Formal Methods and Software Development*
Springer-Verlag Lecture Notes in Computer Science, vols. 185–6, 1985.

[Thatcher, *et al.* 78] J. W. Thatcher, E. G. Wagner, and J. B. Wright
"Data Type Specification: Parameterization and the Power Of Specification Techniques"
Proc. 10th ACM Symposium on Theory of Computing, May 1978, 119–132.

[Wand 79] M. Wand
"Final Algebra Semantics and Data Type Extensions"
J. Computer and System Sciences, vol. 19, no. 1, Aug. 1979, 27–44.

[Wing 83] J. M. Wing
"A Two-Tiered Approach to Specifying Programs"
Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1983, MIT/LCS/TR-299.

[Wing 84] J. M. Wing
"Helping Specifiers Evaluate Their Specifications"
Proc. 2nd ACFET Software Engineering Conference, June 1984, 179–191.

[Wing 85] J. M. Wing
"Writing Larch Interface Language Specifications," submitted for publication.

[Yourdon and Constantine 78] E. Yourdon and L. L. Constantine
Structured Design: Fundamentals of A Discipline of Computer Programs and System Design, 2nd edition, Yourdon Press, 1978.

[Zachary 83] J. L. Zachary
"A Syntax-Directed Specification Editor"
S.M. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Mar. 1983.

[Zave 82] P. Zave
"An Operational Approach to Requirements Specification for Embedded Systems"
IEEE Trans. Software Engineering, vol. 8, no. 3, May 1982, 250–269.

[Zilles] S. N. Zilles
"Abstract Specifications for Data Types"
Technical Report, IBM San Jose Research Laboratory.

Index