

Preliminary Report on The Larch Shared Language *

J. V. Guttag
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

J. J. Horning
Xerox Corporation
3333 Coyote Hill Road
Palo Alto, California 94304

October 1983

ABSTRACT

Each member of the Larch family of formal specification languages has a component derived from a programming language and another component common to all programming languages. We call the former interface languages, and the latter the Larch Shared Language.

This report presents version 1.0 of the Larch Shared Language. It begins with a brief introduction to the Larch Project and the Larch family of languages. The next chapter presents most of the features of the Larch Shared Language and briefly discusses how we expect these features to be used. It should be read before reading either of the remaining two chapters, which are a self-contained reference manual and a set of examples.

Keywords: Algebraic specification, specification language

©J. V. Guttag and J. J. Horning

MIT

*This work was 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, and at the Xerox Palo Alto Research Center by the Computer Science Laboratory.

Table of Contents

Context

The Larch Family of Languages

Status and Plans

An Introduction to the Larch Shared Language

Simple Algebraic Specifications

Getting Richer Theories

Combining Independent Traits

Combining Interacting Traits

Renaming and Exclusion

Assumptions

Consequences

IfThenElse and Equality

Some Further Examples

Larch Shared Language Reference Manual

Structure of Manual

Kernel Syntax

Simple Traits

Consequences and Exemptions

Constrains Clauses

Implicit Signatures and Partial OpForms

Mixfix Operators

Boolean Terms as Equations

External References

Modifications

Implicit Incorporation of Boolean, IfThenElse, and Equality

Semantic Checking

Reference Grammar for Larch Shared Language

Towards a Larch Shared Language Handbook

Preface

Conventions

Basic Properties of Single Operators, Including Binary Relations

Ordering Relations

Group Theory

Simple Numeric Types

Simple Data Structures

Container Properties

Container Classes

Generic Operators on Containers

Nonlinear Structures

Rings, Fields, and Numbers

Lattices

Enumerated Data Types

Display Traits

References

Context

The Larch Family of Languages

The Larch Project is developing tools and techniques intended to aid in the productive use of formal specifications of systems containing computer programs. Many of its premises and goals are discussed in [Guttag, Horning, and Wing 82].

We view a system as consisting of a state and mechanisms for changing and extracting information from that state. We choose to define the information contained in the state without reference to either how that information was created or how it will be used. Our specifications consist of two parts. In one, we specify the properties of values that may appear in system states, and in the second, the program modules that deal with those states.

A major component of the Larch Project is a family of specification languages. Each Larch language has a component particular to a specific programming language and another component common to all programming languages. We call the former *interface languages*, and the latter the *shared language*.

We use the interface languages to specify program modules. Specifications of the interface that one module presents to other modules often rely on notions specific to the programming language, e.g., its denotable values or its exception handling mechanisms. Each interface language deals with what can be observed about the behavior of programs written in a specific programming language. Its simplicity or complexity is a direct consequence of the simplicity or complexity of the observable state and state transformations of that programming language.

The shared language is algebraic. It is used to specify abstractions that are independent of both the program state and the programming language. The operators defined by an algebraic specification appear in specifications written in the interface languages, and in reasoning about such specifications, but they are not directly available to users of programs. The role of shared language specifications is similar to that of abstract models in some other styles of specification.

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

Composability of specifications. We emphasize the incremental construction of specifications from other specifications. The importance of such mechanisms is discussed in [Burstall and Goguen 77]. Larch has mechanisms for building upon and decomposing specifications as well as for combining specifications.

Emphasis on presentation. Reading specifications is an important activity. To assist in this process, we use composition mechanisms defined as operations on specifications, rather than on theories or models.

Interactive and integrated with tools. The Larch languages are designed for interactive use. They are intended to facilitate the interactive construction and incremental checking of specifications. The decision to rely heavily on support tools has influenced our language design in many ways.

Semantic checking. It is all too easy to write specifications with surprising implications. We would like many such specifications to be detectably ill-formed. Extensive checking while specifications are being constructed is an important aspect of our approach. Larch was designed to be used with a powerful theorem prover for semantic checking to supplement the syntactic checks commonly defined for specification languages. We have been influenced here by our experience with Affirm [Musser 80].

Programming language dependencies localized. We feel that it is important to incorporate many programming-language-dependent features into our specification languages, but to isolate this aspect of specifications as much as possible. This prompted us to design a single shared language that could be incorporated into different interface languages in a uniform way.

Shared language based on equations. The shared language has a simple semantic basis taken from algebra. Because of the emphasis on composability, checkability and interaction, however, it differs substantially from the "algebraic" specification languages we have used in the past.

Interface languages based on predicate calculus. Each interface language is based on assertions written in typed first-order predicate calculus with equality, and incorporates programming-language-specific features to deal with constructs such as side effects, exception handling, and iterators. Equality over terms is defined in the shared language; this provides the link between the two parts of a specification.

Status and Plans

We are still in the early phases of the Larch project. In addition to the work described in this report, interface languages for CLU and Mesa have been designed. [Wing 83] contains a detailed description of the semantics of the CLU interface language. The Mesa interface language has not been documented, but we have used it, in conjunction with the shared language, to specify the program level interface to the Cypress data base system. This is the largest specification we have attempted.

A primitive checker for the Shared Language has been implemented [Kownacki 83]. In addition to parsing specifications, this program checks various context sensitive constraints and provides mechanisms for "expanding" assumptions, importations, and inclusions. This checker is an interim tool. We designed our specification language in tandem with an editing and viewing tool. Many language design decisions were influenced by the presumption that specifications would be produced and read interactively using this tool. A first design is complete [Zachary 83], but implementation has yet to begin.

We are in the process of implementing term rewriting software [Forgaard 83], [Lescanne 83] that we hope will provide much of the theorem-proving capability needed for analyzing specifications. The definition of the Larch Shared Language calls for a number of checks for which there can be

no effective procedure. We have what we believe are useful procedures, based on sufficient or necessary (but not both) conditions, for some of these checks, e.g., consistency. We are working on procedures for the others, e.g., checking constrains clauses. This is a difficult task. Diagnostics present a particularly vexing problem: How should relatively complicated theorem-proving procedures report problems to users who are not familiar with either their internal structure or the theory underlying them?

It is always difficult to evaluate a language that has not been extensively used. The Larch Shared Language is especially hard to evaluate because it has been designed for use in an environment that we have not yet built. In addition to the specification of Cypress, we have written a number of small specifications. On the whole, we were pleased by the ease of constructing these specifications in Larch, and with the specifications themselves. While constructing them, we uncovered several errors by inspection; we are encouraged that most of these errors would have been detected automatically by the checks called for in the language definition. It will be some time, however, before we can draw any strong conclusions about the potential utility of Larch in software development.

An Introduction to 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. Consider the following specification for tables that store values in indexed places:

TableSpec: trait

introduces

new: \rightarrow Table
 add: Table, Index, Val \rightarrow Table
 $\# \in \#$: Index, Table \rightarrow Bool
 eval: Table, Index \rightarrow Val
 isEmpty: Table \rightarrow Bool
 size: Table \rightarrow Card

constrains new, add, \in , eval, isEmpty, size so that

for all [*ind*, *indl*: Index, *val*: Val, *t*: Table]

$\text{eval}(\text{add}(t, \text{ind}, \text{val}), \text{indl}) = \text{if } \text{ind} = \text{indl} \text{ then } \text{val} \text{ else } \text{eval}(t, \text{indl})$

$\text{ind} \in \text{new} = \text{false}$

$\text{ind} \in \text{add}(t, \text{indl}, \text{val}) = (\text{ind} = \text{indl}) \mid (\text{ind} \in t)$

$\text{size}(\text{new}) = 0$

$\text{size}(\text{add}(t, \text{ind}, \text{val})) = \text{if } \text{ind} \in t \text{ then } \text{size}(t) \text{ else } \text{size}(t) + 1$

$\text{isEmpty}(t) = (\text{size}(t) = 0)$

This example is similar to a conventional algebraic specification in the style of [Guttag and Horning 80] and [Musser 80]. 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," 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: $\sim(\text{true} = \text{false})$ 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. *Th* is deliberately small, because it is important to prove theorems before a specification is complete, and we wanted to limit the circumstances under which the addition of new operators and equations could invalidate previously proved theorems. 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]), this monotonicity property would have been lost.

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. This is an appropriate limitation when the set of generators for a sort is incomplete. Saying that sort *S* is **generated by** a set of operators, *Ops*, asserts that each term of sort *S* is equal to a term whose outermost operator is in *Ops*. One might, for example, say that the natural numbers are **generated by** 0 and successor and the integers **generated by** 0, successor, and predecessor. **Generated by** adds an inductive rule of inference.

This inductive rule and the clause **Table generated by** [*new*, *add*] can be used to derive theorems such as

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

that would otherwise not be in the theory.

Section 1 allows equations to be derived only by direct equational substitution, not by the absence of inequations. This is an appropriate limitation when the set of observers for a sort is incomplete. 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 rule of inference adds new equations to the theory associated with a trait, thus reducing the number of equivalence classes in the equality relation.

This rule and the clause `Table partitioned by [€, eval]` can be used to derive theorems such as $add(add(t, ind, v), ind1, v) = add(add(t, ind1, v), ind, v)$, that would otherwise not be in the theory.

3. Combining Independent Traits

Our example contains a number of totally unconstrained operators, e.g., `false` and `+`. Such traits are not very useful. The most straightforward thing to do would be 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 in either the importing trait or 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 each of the theories associated with its imported traits. (An extension, $Th1$, of a theory, $Th2$, is conservative if and only if every wff of the language of $Th2$ which is in $Th1$ is also in $Th2$.) Each imported trait can, therefore, be fully understood independently of the context into which it 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 #.rel#: T, T → Bool
constrains .rel so that for all [ t: T ]
    t .rel t = true
```

Symmetric: trait

```
introduces #.rel#: T, T → Bool
constrains .rel so that for all [ t1, t2: T ]
    t1 .rel t2 = t2 .rel t1
```

Transitive: trait

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

Equivalence: trait

includes Reflexive, Symmetric, Transitive

Equivalence has the same associated theory as the less structured trait

Equivalence1: trait

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

Any legal trait importation may be replaced by trait inclusion without either making the trait illegal or changing the associated theory. It does involve the sacrifice of 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 and Exclusion

The specification of Equivalence in the previous section relied heavily on the coincidental use of the operator `.rel` 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 [x for y]

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

Occasionally we wish to eliminate an operator altogether. The phrase

Tr without [op]

stands for the trait Tr without the declaration of op and without each axiom, generated by, and partitioned by in which op appears. We use without to remove an operator either so that we can later add another operator with the same name and signature but different properties or merely because it is superfluous and we want to spare readers the bother of looking at it.

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

```
ArraySpec: trait
  imports IntegerSpec
  includes TableSpec without [ size ]
  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
  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))
```

Notice that in this specification isEmpty is totally unconstrained. In section 7 we discuss a checking mechanism that would call the lack of constraints on isEmpty to the specifier's attention. This would, presumably, provoke him either to add the axioms

```
isEmpty(new) = true
isEmpty(assign(t, ind, val)) = false
```

to his specification, or to add isEmpty to the without clause.

The use of without rather than some sort of hiding mechanism (as in [Burstall and Goguen 81]) may thus involve some extra work for the specifier. In return for this work, users of the specification are spared having to deal with the "hidden" operators, e.g., in proofs that use the specification. This

is consistent with our belief that specifiers should be encouraged to do things that will make life easier for users of their specifications.

The definition of `without` should make it clear that we are indeed operating on the text of traits (presentations) rather than on their associated theories. Consider adding these `isEmpty` axioms to `TableSpec` to form another trait, `TableSpec1`. `TableSpec` and `TableSpec1` have the same associated theories, but

`TableSpec without size`

and

`TableSpec1 without size`

have rather different associated theories—in the latter, `isEmpty` is fully defined.

A final point raised by the examples of this section is the importance of distinguishing 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. Assumptions

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

`MultiSetSpec: trait`

`introduces`

`{}: → MultiSet`

`insert: MultiSet, Elem → MultiSet`

`delete: MultiSet, Elem → MultiSet`

`# ∈ #: MultiSet, Elem → Bool`

`constrains {}, insert, delete, ∈ so that`

`MultiSet generated by [{}, insert]`

`MultiSet partitioned by [delete, ∈]`

`for all [m: MultiSet, e, el: Elem]`

`e ∈ {} = false`

`e ∈ insert(m, el) = (e = el) | (e ∈ m)`

`delete({}, e) = {}`

`delete(insert(m, e), el) =`

`if e = el then m else insert(delete(m, el), e)`

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

`IntMultiSet: trait`

`imports IntegerSpec`

`includes MultiSetSpec with [Integer for Elem]`

The interactions between `MultiSetSpec` and `IntegerSpec` are very limited. Nothing in `MultiSetSpec` places any constraints on the meaning of the operators that occur in `IntegerSpec`, e.g., `0`, `+`, and `<`. Consider, however, extending `MultiSetSpec` to `MultiSetSpec1` by adding an operator `rangeCount`,

```
MultiSetSpec1: trait
  imports MultiSetSpec, Cardinal
  introduces
    rangeCount: MultiSet, Elem, Elem → Integer
    # < #: Elem, Elem → Bool
  constrains rangeCount so that for all [ e1, e2, e3: Elem, m: MultiSet ]
    rangeCount({}, e1, e2) = 0
    rangeCount(insert(m, e3), e1, e2) =
      rangeCount(m, e1, e2) + (if (e1 < e3) & (e3 < e2) then 1 else 0)
```

`MultiSetSpec1` places no constraints on 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 could specify such a restriction by adding to `MultiSetSpec1` the assumption (`Ordered` is defined in the Handbook section, on page 36):

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

In constructing the theory associated with `MultiSetSpec1`, the assumption would be treated as if `Ordered with [Elem for T]` had been included. This could be used to derive various properties of `MultiSetSpec1`, e.g., that `rangeCount` is monotonic in its last argument.

Whenever the augmented `MultiSetSpec1` is imported or included in another trait, however, the assumption will have to be discharged. In

```
IntMultiSet1: trait
  includes MultiSetSpec1 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. Consequences

We have now looked at those parts of the Larch Shared Language that determine the theory associated with a valid 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 the augmented **MultiSetSpec1**,

implies for all [*m*: MultiSet, *e1*, *e2*, *e3*: Elem]

$(e2 < e3) \Rightarrow (\text{rangeCount}(m, e1, e2) \leq \text{rangeCount}(m, e1, e3))$

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*]. This asserts that each term is provably equal to a term that does not contain operators in *Ops*. (We do not require this for terms containing variables of sorts appearing in **generated by** clauses.) **Converts** is used to say that the specification adequately defines a collection of operators.

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 each term containing **isEmpty**, such as **isEmpty(new)** or **isEmpty(add(new), ind, val)**, is equal to another term that does 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 would present a problem if we did not wish to add an axiom to resolve this incompleteness. We therefore provide a mechanism to allow specifiers to indicate that the unconvertibility of certain terms is acceptable. If **TableSpec** were modified to include

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

the checking associated with the **converts** would now require that the theory associated with **TableSpec** must contain either

an equation, $t = t1$, where $t1$ has no occurrences of **isEmpty** or **eval**, or

an equation $t' = t1$, where t' is a subterm of t , and $t1$ 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. One use of **converts** is to allow the specification checker to notice unintended effects of **without**. As suggested in section 6, the failure of **ArraySpec** to fulfill the **converts** inherited from **TableSpec** would trigger error messages of the form:

isEmpty(new) not convertible

isEmpty(assign(*t*, *ind*, *val*)) not convertible.

8. IfThenElse and Equality

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

Whenever a term of the form `if b then t1 else t2` occurs in a trait we replace the infix symbol `if then else` by the prefix symbol `ifThenElse`. If `t1` and `t2` 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 `t1 = t2` occurs in a trait, if `t1` and `t2` are of the same sort, `T1`, we append the trait `Equality` with `[T1 for T]` to the consequences of the enclosing trait.

Specifications of these traits are:

`IfThenElse`: trait

```
introduces ifThenElse: Bool, T, T → T
constrains ifThenElse so that for all [ t1, t2: T ]
    ifThenElse(true, t1, t2) = t1
    ifThenElse(false, t1, t2) = t2
implies converts [ ifThenElse ]
```

`Equality`: trait

```
includes Equivalence with [ = for .rel ]
constrains = so that T partitioned by [ = ].
```

9. Some Further Examples

The following series of examples is adapted from the Handbook chapter. We include them here to illustrate some ways in which the facilities introduced above can be used. In reading these specifications, keep in mind that they are not themselves ends, but rather means to write interface specifications.

Our first example is an abstraction of those data structures that “contain” elements, e.g., `Set`, `Bag`, `Queue`, `Stack`. We have found it useful both as a starting point for specifications of various kinds of containers, and as an assumption for generic operations. The crucial part of the trait is the `generated by`. It indicates that any term of sort `C` is equal to some term in which `new` and `insert` are the only operators with range `C`—even if this trait is included in one that introduces additional operators that return values of sort `C`. This means that any theorems proved by induction over `new` and `insert` will remain valid.

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

The next example incorporates `Container` as an assumption. Notice that it constrains `new` and `insert` as well as the operator it introduces, `isEmpty`. The `converts` indicates that this trait contains

enough axioms to adequately specify isEmpty. Because of the **generated by**, this can be proved by induction over terms of sort C, using new as the basis and 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 ]
```

The next two examples assume Container. The **exempts** indicate that should these traits be included into a trait that claims the convertibility of next or rest, that trait needn't convert the terms next(new) or rest(new).

```
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)
```

The next example specifies properties common to various data structures such as stacks, queues, priority queues, sequences, and vectors. It augments Container by combining it with IsEmpty, Next, and Rest. The **partitioned by** indicates that next, rest, and isEmpty are sufficient to define equality over terms of sort C. Since we have little information about next and rest, the **partitioned by** does not yet add much to the associated theory.

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

The next example 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.

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) $\leq e$ then next(q) else e

rest(insert(q, e)) =

if isEmpty(q) then new

else if next(q) $\leq e$ then insert(rest(q), e) else q

implies converts [next, rest, isEmpty]

In a trait, such as PriorityQueue, that defines an “abstract data type” there will generally be a distinguished sort (C in this case) 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.

The next example illustrates a specialization of Container that does not satisfy Enumerable. It augments Container by combining it with IsEmpty and Cardinal, and introducing two new operators. Notice that we include Container, because we intend to constrain operators inherited from it, but import IsEmpty and Cardinal, because we do not intend to constrain any operator inherited from them. Constrains C is a shorthand for a constrains clause listing all the operators whose signature includes C . The partitioned by indicates that count alone is sufficient to distinguish unequal terms of sort C . Converts [isEmpty, count, delete] is a stronger assertion than the combination of an explicit converts [count, delete] with the inherited converts [isEmpty].

MultiSet: trait

assumes Equality with [Elem for T]

imports IsEmpty, Cardinal

includes Container with [empty for new]

introduces count: Elem, $C \rightarrow \text{Bool}$

delete: Elem, $C \rightarrow C$

constrains C so that

C partitioned by [count]

for all [$c: C, e1, e2: E$]

count(empty, $e1$) = 0

count(insert($c, e1$), $e2$) = count($c, e2$) + (if $e1 = e2$ then 1 else 0)

delete(empty, $e1$) = empty

delete(insert($c, e1$), $e2$) =

if $e1 = e2$ then c else insert(delete($c, e2$), $e1$)

implies converts [isEmpty, count, delete]

The next example specifies a generic operator. It uses `Enumerable` as an assumption to delimit the applicability of this operator to 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 O, C for T ]

```

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

Larch Shared Language Reference Manual

0. Structure of Manual

In section 1 we present a grammar for the kernel subset of the Larch Shared Language.

In section 2 we define the context sensitive checking and the theory associated with each specification written in the kernel subset.

In section 3 we extend the kernel subset by introducing mechanisms for specifying intended consequences of a specification written in the kernel subset.

In sections 4-10 we define successive extensions of the language. We modify the grammar to introduce additional aspects of the language and describe any additional context sensitive checking required. We also provide a translation from the newly extended language to the previously defined subset. The result of this translation is subjected to all the applicable checking. The theory associated with any specification written in the full language is the same as the theory associated with its translation.

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 as well as the context sensitive checks described earlier.

Finally, section 12 collects the reference grammar for the entire language.

1. Kernel Syntax

1.1. Syntactic conventions

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

1.2. Grammar

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

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

2. Simple Traits

2.1. Context sensitive checking

simpleTrait:

- = The sets of *varId*'s, *sortId*'s and *opId*'s appearing in a *trait* must be disjoint.
- Every *sortId* appearing anywhere in a *simpleTrait* must appear in its *opPart*.
- Every *sortedOp* appearing anywhere in a *simpleTrait* must appear in its *opPart*.

opDcl:

Each *opForm* must have the same number of #'s as the number of occurrences of *sortId*'s in the 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 *term*'s 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**, .

2.2. Associated theory

We associate a theory with each *trait*. This section defines the theory associated with a *simpleTrait*.

A theory is a subset of the language:

wff ::= *term* = *term*

- | "propositional formula"
- | "first order quantified (with sorts) formula"

We adopt the conventional meanings of the equality symbol (=), the propositional connectives (&, |, ~, ⇒, ...), and the quantifiers (∀ and ∃).

The subset of *wff* that is the theory, call it *Th*, associated with a *simpleTrait* is defined by:

Axioms: Each *equation*, universally quantified by the *varDcl*'s of its containing *axioms*, is in *Th*.

Inequation: $\sim(\text{true}:\rightarrow\text{Bool} = \text{false}:\rightarrow\text{Bool})$ 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 [*op*₁, ..., *op*_{*n*}], and *P*(*s*) is a *wff* with a free variable, *s*, of sort *S*, *Th* contains the *wff*

∀[*s*: *S*] *P*(*s*)

if for each *op*_{*i*} in [*op*₁, ..., *op*_{*n*}]

$Q_i \Rightarrow P(\text{op}_i(x_1, \dots, x_k))$ 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 [*op*₁, ..., *op*_{*n*}], *Th* contains the *wff*

∀[*s*₁, *s*₂: *S*] (*Q* ⇒ *s*₁ = *s*₂)

where *Q* is the conjunction, for each *op*_{*i*} in [*op*₁, ..., *op*_{*n*}] and each *j* such that the *j*th argument of *op*_{*i*} is of sort *S*, of

∀[*x*₁: *S*₁, ..., *x*_{*k*}: *S*_{*k*}] (*Subst*(*op*_{*i*}, *j*, *t*₁) = *Subst*(*op*_{*i*}, *j*, *t*₂)), where

*S*₁, ..., *S*_{*k*} is the domain of *op*_{*i*}, and

Subst(*op*, *j*, *t*) is *op*(*x*₁, ..., *x*_{*k*}) with *t* substituted for *x*_{*j*}.

3. Consequences and Exemptions

Exempts and *consequences* affect only the checking (see section 11.5) 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*,

```

3.1. 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 2.1.

4. Constrains Clauses

Constrains clauses affect only the checking (see section 11.4), not the theory. We add to the grammar the productions:

```

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

```

4.1. Translation

constrains:

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* { \rightarrow *range* }

5.1. Context sensitive checking

There must be a unique mapping from occurrences of *sortedOp*'s to *opDcl*'s of the *traitBody* such that the translation described in section 5.2. produces a legal *traitBody* and for each *sortedOp*, *opDcl* pair:

The *opId*'s 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 \rightarrow *range*, it is the same as the *range* of the *opDcl*.

5.2. 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* '

6.1. 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 t₁ else t₂* into a term of the form *ifThenElse(b, t₁, t₂)*.

secondary:

Translate each *secondary* containing *opSym*'s 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 *primary*'s.

primary:

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

7. Boolean Terms as Equations

It is convenient to use terms of sort *Bool* as axioms. We add to the grammar the production:
equation ::= term

7.1. Context sensitive checking

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

7.2. 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

```

8.1. 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*.

8.2. 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 *propPart*'s and *E*'s *propPart*'s,
- An *exempts* containing *T*'s *exemptTerms* and *E*'s *exemptTerms*, and
- A *consequences* containing the *props* of
 - T*'s *conseqProps*,
 - the *propParts* of the translations of the *traitRef*'s in *T*'s *conseqProps*, and
 - E*'s *consequences*.

9. Modifications

We add to the grammar the productions:

traitRef ::= *traitId* {*exclusion*} {*renaming*}
exclusion ::= **without** [*oldOp*^{*},]
renaming ::= **with** [(*sortRename* | *opRename*)^{*},]
sortRename ::= *sortId* for *oldSort*
oldSort ::= *sortId*
opRename ::= *opId* for *oldOp*
oldOp ::= *sortedOp*

9.1. 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 5.1),

If the *oldOp* includes *domain*, it is the same as the *domain* of the *opDcl*.

If the *oldOp* includes \rightarrow *range*, it is the same as the *range* of the *opDcl*.

9.2. Translation

The translation of the *trait* labeled by the *traitId* of the *traitRef* is modified by applying first the *exclusion*, then the *opRename*'s, and finally the *sortRename*'s:

For each *oldOp* in the *exclusion*, delete each *bylist*, *equation*, and *term* containing the *opDcl* to which it maps and then delete all remaining occurrences of that *opDcl*.

Then, 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*.

Finally, 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.

10.1. 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 t₁ else t₂* in which *t₁* and *t₂* 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 *t₁ = t₂* in which *t₁* and *t₂* have the same sort, T1.

10.2. Built-in traits

Boolean: trait

introduces
 true: → Bool
 false: → Bool
 ~#: Bool → Bool
 #&#: Bool, Bool → Bool
 #|#: Bool, Bool → Bool
 #⇒#: Bool, Bool → Bool
 #.equal#: 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 .equal b) = b
 (false .equal b) = ~b

implies converts [~, &, |, ⇒, .equal]

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 $\# = \# : T, T \rightarrow \text{Bool}$

asserts T partitioned by [=]

for all [x, y, z: T]

$(x = x)$

$(x = y) = (y = x)$

$((x = y) \ \& \ (y = z)) \Rightarrow (x = z)$

11. Semantic Checking

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

11.1. Consistency

A *traitBody* is *consistent* if its associated theory does not contain the equation
 $\text{true} : \rightarrow \text{Bool} = \text{false} : \rightarrow \text{Bool}$

11.2. 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 $R(T)$.

11.3. 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* of T2, W is in the theory associated with T1 if and only if it is in the theory associated with T2.

11.4. 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 *sortedOp*'s 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 *ops* in *sortedOp**.

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

11.5. 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.

Note that an *exclusion*, but not a *renaming*, can invalidate a consequence that has been locally checked.

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 $t = u$,

where *u* contains no *sortedOp* appearing in *C*'s *sortedOp**, or

contain an equation $t' = u$,

where *t'* is a subterm of *t*, and *u* is an instantiation of a *term* appearing in an *exempts* of the containing *trait*.

12. Reference Grammar for The Larch Shared Language

trait	::= <i>traitId</i> : trait <i>traitBody</i> { <i>consequences</i> } { <i>exempts</i> }
traitBody	::= <i>externals</i> <i>simpleTrait</i>
externals	::= { <i>assumes</i> } { <i>imports</i> } { <i>includes</i> }
assumes	::= assumes <i>traitRef</i> *
imports	::= imports <i>traitRef</i> *
includes	::= includes <i>traitRef</i> *
traitRef	::= <i>traitId</i> { <i>exclusion</i> } { <i>renaming</i> }
exclusion	::= without [<i>oldOp</i> * ,]
renaming	::= with [(<i>sortRename</i> <i>opRename</i>)*,]
sortRename	::= <i>sortId</i> for <i>oldSort</i>
oldSort	::= <i>sortId</i>
opRename	::= <i>opId</i> for <i>oldOp</i>
oldOp	::= <i>sortedOp</i>
sortedOp	::= <i>opDcl</i> <i>opId</i> { \rightarrow <i>range</i> }
simpleTrait	::= { <i>opPart</i> } <i>propPart</i> *
opPart	::= introduces <i>opDcl</i> *
opDcl	::= <i>opId</i> : <i>signature</i>
signature	::= <i>domain</i> \rightarrow <i>range</i>
domain	::= <i>sortId</i> *
range	::= <i>sortId</i>
propPart	::= (asserts constrains) <i>props</i>
constrains	::= constrains (<i>sortId</i> <i>sortedOp</i> * ,) so that
props	::= <i>generators</i> * <i>partitions</i> * <i>axioms</i> *
generators	::= <i>sortId</i> generated <i>bylist</i> *
partitions	::= <i>sortId</i> partitioned <i>bylist</i> *
bylist	::= by [<i>sortedOp</i> * ,]
axioms	::= for all [<i>varDcl</i> * ,] <i>equation</i> *
varDcl	::= <i>varId</i> * , : <i>sortId</i>
equation	::= <i>term</i> { = <i>term</i> }
term	::= <i>secondary</i> if <i>secondary</i> then <i>secondary</i> else <i>term</i>
secondary	::= { <i>opSym</i> } <i>primary</i> (<i>opSym</i> <i>primary</i>)* { <i>opSym</i> }
primary	::= <i>sortedOp</i> { '(<i>term</i> * , ') } <i>varId</i> '(<i>term</i> ')
opId	::= <i>alphaNumeric</i> + <i>opForm</i>
opForm	::= { # } <i>opSym</i> (# <i>opSym</i>)* { # }
opSym	::= <i>specialChar</i> + . <i>alphaNumeric</i> +
traitId	::= <i>alphaNumeric</i> +
sortId	::= <i>alphaNumeric</i> +
varId	::= <i>alphaNumeric</i> +
consequences	::= implies <i>conseqProps</i> { <i>converts</i> }
conseqProps	::= <i>traitRef</i> * , <i>props</i>
converts	::= converts <i>conversion</i> * ,
conversion	::= [<i>sortedOp</i> * ,]
exempts	::= exempts <i>exemptTerms</i> *
exemptTerms	::= { for all [<i>varDcl</i> * ,] } <i>term</i> * ,

Towards A Larch Shared Language Handbook

Contents

Basic properties of single operators, including binary relations

Associative, Commutative, Idempotent, Relation, TotalRelation, Reflexive, Irreflexive, Transitive, ReflexiveTransitive, Symmetric, Antisymmetric, Equivalence

Ordering relations

PartialOrder, TotalOrder, OrderEquivalence, OrderEquality, PartialOrderWithEquality, TotalOrderWithEquality, DerivedOrders, PartiallyOrdered, Ordered

Group theory

LeftIdentity, RightIdentity, Identity, LeftInverse, RightInverse, Inverse, Abelian, Semigroup, Monoid, Group, AbelianSemigroup, AbelianMonoid, AbelianGroup, Distributive

Simple numeric types

Ordinal, Cardinal, Cardinal2

Simple data structures

Pair, Triple, FiniteMapping

Container properties

Container, Singleton, IsEmpty, Size, AdditiveSize, Join, ElementEquality, Member, ElemCount, Delete, Containment, Next, Rest, Remainder, Index

Container classes

SetBasics, BagBasics, CollectionExtensions, SetIntersection, Set, Bag, Enumerable, InsertionOrdered, Stack, Queue, Dequeue, Sequence, SubSequence, String, PriorityQueue

Generic operators on containers

CoerceContainer, Reduce, SomePass, AllPass, Sift, PairwiseExtension, PointwiseImage

Nonlinear structures

BinaryTree, BasicGraph, Connectivity, Graph

Rings, fields, and numbers

Ring, RingWithUnit, InfixInverse, Integer, Field, Rational

Lattices

ExtremalBound, Semilattice, Lattice

Enumerated data types

Enumerated, Rainbow, Character

Display traits

Coordinate, Illumination, Boundary, Transform, Displayable, Picture, Contents, Component,
ComponentCoercion, View, Display

Preface

This collection of traits is a companion to the Larch Shared Language Reference Manual. We hope that it will serve three distinct purposes:

- 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, and
- Help people to better understand the Larch Shared Language by providing a set of illustrative examples.

In line with our first goal, 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. This is particularly evident in the sections titled "Container properties" and "Container classes." The traits in these sections are smaller and more numerous than is typical in "from scratch" specifications. This sometimes leads to a somewhat overstructured appearance.

In line with our second goal, in addition to traits that we expect to be directly incorporated in specifications, we have included a number of traits intended primarily as patterns. The section titled "Generic operators on containers" contains several such traits. Because of the arity of the operators, it will frequently be awkward to incorporate these traits.

In line with our third goal we have stressed 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. The section on Display traits is more typical in this respect.

The support tools envisioned for Larch are not yet available. Transcriptions of traits in this chapter have been mechanically checked for some properties; some errors may not have been detected and some transcription errors may have crept in. They should be given the same sort of credence as carefully written programs that have not been checked by a compiler.

Comments on the clarity of these specifications and on their "correctness" (relative to generally accepted definitions of the names used) are welcome. We also solicit contributions of further widely useful traits—either accompanied by specifications, or as challenges to specifiers.

Conventions

- If a generic trait constrains only one interesting sort, the identifier T is used to denote it.
- If a trait constrains a "containing" sort and an "element" sort, the identifiers C and E are used.
- If a trait constrains a single binary operation, the infix symbol #O# is used.
- If a trait constrains a single binary relation, the infix identifier #R# is used.
- If there would be no information in a constrains (e.g., because there is only one operator), asserts is used.

Basic Properties of Single Operators, Including Binary Relations**Associative: trait****introduces** # \circ #: $T, T \rightarrow T$ **asserts for all** [$x, y, z: T$]

$$(x \circ y) \circ z = x \circ (y \circ z)$$

Commutative: trait**introduces** # \circ #: $T, T \rightarrow \text{Range}$ **asserts for all** [$x, y: T$]

$$x \circ y = y \circ x$$

Idempotent: trait**introduces** op: $T \rightarrow T$ **asserts for all** [$x: T$]

$$\text{op}(\text{op}(x)) = \text{op}(x)$$

Relation: trait**introduces** # \oplus #: $T, T \rightarrow \text{Bool}$ **TotalRelation: trait****includes** Relation**asserts for all** [$x, y: T$]

$$(x \oplus y) \mid (y \oplus x)$$

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

$$x \oplus x$$

Irreflexive: trait**includes** Relation**asserts for all** [$x: T$]

$$\sim(x \oplus x)$$

Transitive: trait**includes** Relation**asserts for all** [$x, y, z: T$]

$$((x \oplus y) \ \& \ (y \oplus z)) \Rightarrow (x \oplus z)$$

ReflexiveTransitive: trait**includes** Reflexive, Transitive**Symmetric: trait****includes** Relation**asserts for all** [$x, y: T$]**implies** Commutative with [\oplus for \circ , Bool for Range]

$$(x \oplus y) = (y \oplus x)$$

Antisymmetric: trait**includes** Relation**asserts for all** [$x, y: T$]**implies** Irreflexive

$$\sim((x \oplus y) \ \& \ (y \oplus x))$$

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

Ordering Relations

PartialOrder: trait

imports ReflexiveTransitive with [\leq for \otimes]

TotalOrder: trait

includes PartialOrder, TotalRelation with [\leq for \otimes]

OrderEquivalence: trait

assumes PartialOrder

introduces #.eq#: T, T \rightarrow Bool

constrains .eq so that for all [x, y: T] (x.eq y) = (x \leq y) & (y \leq x)

implies Equivalence

converts [.eq]

OrderEquality: trait

assumes PartialOrder

includes OrderEquivalence with [= for .eq], Equality

PartialOrderWithEquality: trait

includes PartialOrder, OrderEquality

TotalOrderWithEquality: trait

includes TotalOrder, OrderEquality

DerivedOrders: trait

assumes PartialOrder

introduces

#<#: T, T \rightarrow Bool

\geq #: T, T \rightarrow Bool

#>#: T, T \rightarrow Bool

constrains < so that for all [x, y: T] (x < y) = ((x \leq y) & (\sim (y \leq x)))

constrains \geq so that for all [x, y: T] (x \geq y) = (y \leq x)

constrains > so that for all [x, y: T] (x > y) = (y < x)

implies Transitive with [< for \otimes],

Transitive with [> for \otimes],

Antisymmetric with [< for \otimes],

Antisymmetric with [> for \otimes],

PartialOrder with [\geq for \leq]

converts [<, \geq , >]

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]

Group Theory**LeftIdentity: trait****introduces** $\# \circ \# : T, T \rightarrow T$ $\text{unit} : \rightarrow T$ **asserts for all [x: T]** $\text{unit} \circ x = x$ **RightIdentity: trait****introduces** $\# \circ \# : T, T \rightarrow T$ $\text{unit} : \rightarrow T$ **asserts for all [x: T]** $x \circ \text{unit} = x$ **Identity: trait includes LeftIdentity, RightIdentity****LeftInverse: trait****assumes LeftIdentity****introduces inv: T → T****asserts for all [x: T]** $\text{inv}(x) \circ x = \text{unit}$ **RightInverse: trait****assumes RightIdentity****introduces inv: T → T****asserts for all [x: T]** $x \circ \text{inv}(x) = \text{unit}$ **Inverse: trait****assumes Identity****includes LeftInverse, RightInverse****Abelian: trait imports Commutative with [T for Range]****Semigroup: trait includes Associative, Equality****Monoid: trait includes Semigroup, LeftIdentity****Group: trait****includes Monoid, LeftInverse****implies RightIdentity, RightInverse****AbelianSemigroup: trait includes Abelian, Semigroup****AbelianMonoid: trait****includes Abelian, Monoid****implies RightIdentity****AbelianGroup: trait includes Abelian, Group****Distributive: trait****introduces** $\# + \# : T, T \rightarrow T$ $\# * \# : T, T \rightarrow T$ **asserts for all [x, y, z: T]** $x*(y + z) = (x*y) + (x*z)$ $(y + z)*x = (y*x) + (z*x)$

Simple Numeric Types

Ordinal: trait

includes PartialOrder with [= for .eq, Ord for T],
 OrderEquivalence with [= for .eq, Ord for T]
introduces
 first: \rightarrow Ord
 succ: Ord \rightarrow Ord
asserts Ord generated by [first, succ]
 Ord partitioned by [\leq]
 for all [x, y: Ord]
 first \leq x
 $\sim(\text{succ}(x) \leq \text{first})$
 $\text{succ}(x) \leq \text{succ}(y) = x \leq y$
implies TotalOrderWithEquality with [Ord for T]
converts [\leq , =]

Cardinal: trait

imports Ordinal with [0 for first, Card for Ord]
includes DerivedOrders with [Card for T]
introduces
 1: \rightarrow Card
 # + #: Card, Card \rightarrow Card
 # * #: Card, Card \rightarrow Card
 # \ominus #: Card, Card \rightarrow Card
constrains 1 so that 1 = succ(0)
constrains +, * so that for all [x, y: Card]
 $x + 0 = x$
 $x + \text{succ}(y) = \text{succ}(x + y)$
 $x * 0 = 0$
 $x * \text{succ}(y) = x + (x * y)$
constrains \ominus so that for all [x, y: Card]
 $0 \ominus x = 0$
 $x \ominus 0 = x$
 $\text{succ}(x) \ominus \text{succ}(y) = x \ominus y$
implies Cardinal2
 Card generated by [1, +, \ominus]
 Card partitioned by [\geq], by [=], by [<], by [>]
 for all [x, y: Card] $x \leq y = ((x \ominus y) = 0)$
converts [1, \ominus , +, *, =, \leq , \geq , <, >]

```

Cardinal2: trait                                % Alternate definition for comparison
includes AbelianMonoid with [ + for O, 0 for unit, Card for T ],
          AbelianMonoid with [ * for O, 1 for unit, Card for T ],
          Distributive with [ Card for T ],
          Ordered with [ Card for T ]
introduces
          #  $\ominus$  #: Card, Card  $\rightarrow$  Card
          succ: Card  $\rightarrow$  Card
asserts Card generated by [ 0, 1, + ]
          for all [ x, y: Card ]
              x < (x + 1)
              (x + y)  $\ominus$  y = x
              0  $\ominus$  x = 0
              succ(x) = x + 1
implies Cardinal

```

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, il: Index, e: E ]
    bind(c, il, e)[i] = if i = il then e else c[i]
    ~defined(new, i)
    defined(bind(c, il, e), i) = (i = il) | defined(c, i)
implies converts [ #[#], defined ]
exempts for all [ i: Index ] new[i]

```


Container Properties**Container: trait**

introduces
 new: $\rightarrow C$
 insert: $C, E \rightarrow C$
 asserts C generated by [new, insert]

Singleton: trait

assumes Container
 introduces singleton: $E \rightarrow C$
 constrains singleton so that for all [$e: E$]
 singleton(e) = insert(new, e)
 implies converts [singleton]

IsEmpty: trait

assumes Container
 introduces isEmpty: $C \rightarrow \text{Bool}$
 asserts for all [$c: C, e: E$]
 isEmpty(new)
 ~isEmpty(insert(c, e))
 implies converts [isEmpty]

Size: trait

assumes Container
 imports Cardinal
 introduces size: $C \rightarrow \text{Card}$
 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#: $C, C \rightarrow C$
 constrains join so that for all [$c, cl: C, e: E$]
 c .join new = c
 c .join insert(cl, e) = insert(c .join cl, e)
 implies converts [.join]

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

assumes Container, ElementEquality
 introduces #∈#: $E, C \rightarrow \text{Bool}$
 constrains ∈, insert so that for all [$c: C, e, el: E$]
 ~($e \in$ new)
 $e \in$ insert(c, el) = ($e = el$) | ($e \in c$)
 implies converts [∈]

ElemCount: trait

assumes Container, ElementEquality
imports Cardinal
introduces count: $C, E \rightarrow \text{Card}$
constrains count, insert so that for all [$e, el: E, c: C$]
 count(new, e) = 0
 count(insert(c, e), el) = count(c, e) + (if $e = el$ then 1 else 0)
implies converts [count]

Delete: trait

assumes Container
introduces delete: $C, E \rightarrow C$
constrains delete so that for all [$e: E$] delete(new, e) = new

Containment: trait

assumes Container
includes PartiallyOrdered with [C for $<$, \supset for $>$, \subseteq for \leq , \supseteq for \geq , C for T]
constrains C so that for all [$e: E, c: C$] $c \subseteq \text{insert}(c, e)$
implies for all [$c: C$] new $\subseteq c$

Next: trait

assumes Container
introduces next: $C \rightarrow E$
constrains next, insert so that for all [$e: E$] next(insert(new, e)) = e
exempts next(new)

Rest: trait

assumes Container
introduces rest: $C \rightarrow C$
constrains rest, insert so that for all [$e: E$] rest(insert(new, e)) = new
exempts rest(new)

Remainder: trait

assumes Container, Rest
imports Cardinal
introduces remainder: $C, \text{Card} \rightarrow C$
constrains remainder so that for all [$c: C, i: \text{Card}$]
 remainder($c, 0$) = c
 remainder($c, i + 1$) = remainder(rest(c), i)
implies converts [remainder]

Index: trait

assumes Container, Next, Rest
imports Cardinal
introduces $\#[\#]$: $C, \text{Card} \rightarrow E$
constrains $\#[\#]$ so that for all [$c: C, i: \text{Card}$]
 $c[1]$ = next(c)
 $c[(i + 1)]$ = rest(c)[i]
implies converts [$\#[\#]$]
exempts for all [$c: C$] $c[0]$

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, el: E]
 size(insert(s, e)) = size(s) + (if e ∈ s then 0 else 1)
 el ∈ delete(s, e) = (el ∈ s) & (~(e = el))
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, el: E]
 count(delete(b, e), el) = count(b, el) - (if e = el 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 C so that for all [s, sl: C, e, el: E]
 e ∈ (s ∩ sl) = (e ∈ s) & (e ∈ sl)
converts [∩]

Set: trait

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

Bag: trait

assumes ElementEquality
imports BagBasics
includes CollectionExtensions
implies Abelian with [U for O, 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]

InsertionOrdered: trait

% For assuming "Stack or Queue"

includes Enumerable
introduces isFIFO: → Bool
constrains next, rest, insert so that for all [c: C, e: E]
 next(insert(c, e)) = if isEmpty(c) | isFIFO then e else next(c)
 rest(insert(c, e)) = if isEmpty(c) | isFIFO then c else insert(rest(c), e)
implies converts [next, rest]

Stack: trait

includes InsertionOrdered with [push for insert, top for next, pop for rest,
 true for isFIFO]
implies for all [stk: C, e: E]
 top(push(stk, e)) = e
 pop(push(stk, e)) = stk

Queue: trait

includes InsertionOrdered with [first for next, false for isFIFO]
implies 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)

Dequeue: 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, el: E]
 insert(new, e) = enter(new, e)
 insert(enter(c, e), el) = enter(insert(c, el), 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]

String: trait

imports Character

includes Sequence with [length for size, Char for E]

PriorityQueue: trait

assumes TotalOrder with [E for T]

includes Enumerable

constrains next, rest, insert so that for all [$q: C, e: E$]

$\text{next}(\text{insert}(q, e)) = \text{if } \text{isEmpty}(q) \text{ then } e$
 $\text{else if } \text{next}(q) \leq e \text{ then } \text{next}(q) \text{ else } e$

$\text{rest}(\text{insert}(q, e)) = \text{if } \text{isEmpty}(q) \text{ then new}$

$\text{else if } \text{next}(q) \leq e \text{ then } \text{insert}(\text{rest}(q), e) \text{ else } q$

implies converts [next, rest, isEmpty]

Generic Operators on Containers

CoerceContainer: trait

assumes Container with [DC for C],

Container with [RC for C]

introduces coerce: DC \rightarrow RC

constrains coerce so that for all [$dc: DC, e: E$]

$\text{coerce}(\text{new}) = \text{new}$

$\text{coerce}(\text{insert}(dc, e)) = \text{insert}(\text{coerce}(dc), e)$

implies converts [coerce]

Reduce: trait

assumes Enumerable,

RightIdentity with [E for T],

Associative with [E for T]

introduces reduce: C \rightarrow E

constrains reduce so that for all [$c: C$]

$\text{reduce}(c) = \text{if } \text{isEmpty}(c) \text{ then unit else } \text{next}(c) \circ \text{reduce}(\text{rest}(c))$

implies converts [reduce]

SomePass: trait

assumes Container

introduces

test: E, T \rightarrow Bool

somePass: C, T \rightarrow Bool

constrains somePass so that for all [$c: C, e: E, t: T$]

$\sim \text{somePass}(\text{new}, t)$

$\text{somePass}(\text{insert}(c, e), t) = \text{test}(e, t) \mid \text{somePass}(c, t)$

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** InsertionOrdered**introduces**

extOp: C, C → C

elemOp: E, E → E

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)

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]

Nonlinear Structures

BinaryTree: trait

imports Cardinal

introduces

$\langle \# \rangle: E \rightarrow C$
 $\langle \#, \# \rangle: C, C \rightarrow C$
 $\#.left: C \rightarrow C$
 $\#.right: C \rightarrow C$
 $size: C \rightarrow \text{Card}$
 $isLeaf: C \rightarrow \text{Bool}$
 $content: C \rightarrow E$

constrains C so that

C generated by [$\langle \# \rangle, \langle \#, \# \rangle$]
C partitioned by [$\#.left, \#.right, content, isLeaf$]
for all [$tl, tr: C, e: E$]
 $(\langle tl, tr \rangle).left = tl$
 $(\langle tl, tr \rangle).right = tr$
 $size(\langle e \rangle) = 1$
 $size(\langle tl, tr \rangle) = size(tl) + size(tr)$
 $isLeaf(\langle e \rangle)$
 $\sim isLeaf(\langle tl, tr \rangle)$
 $content(\langle e \rangle) = e$

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

$empty: \rightarrow \text{Graph}$
 $addNode: \text{Graph}, \text{Node} \rightarrow \text{Graph}$
 $addEdge: \text{Graph}, \text{Edge} \rightarrow \text{Graph}$
 $nodes: \text{Graph} \rightarrow \text{NodeSet}$
 $adj: \text{Node}, \text{Graph} \rightarrow \text{NodeSet}$

constrains Graph so that

Graph generated by [$empty, addNode, addEdge$]
Graph partitioned by [$nodes, adj$]
for all [$g: \text{Graph}, e: \text{Edge}, n, nl: \text{Node}$]
 $nodes(empty) = \{\}$
 $nodes(addNode(g, n)) = insert(nodes(g), n)$
 $nodes(addEdge(g, e)) = insert(insert(nodes(g), e.first), e.second)$
 $adj(n, empty) = \{\}$
 $adj(n, addNode(g, nl)) = adj(n, g)$
 $adj(n, addEdge(g, e)) =$
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]

Rings, Fields, and Numbers**Ring: trait**

includes AbelianGroup with [+ for \odot , 0 for unit, - # for inv],
 Semigroup with [* for \odot],
 Distributive

RingWithUnit: trait

includes Ring, Identity with [* for \odot , 1 for unit]

InfixInverse: trait

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

Integer: trait

includes RingWithUnit with [Int for T],
 Ordered with [Int for T],
 InfixInverse with [+ for \odot , - # for inv, - for \oslash , Int for T]
 asserts Int generated by [1, +, - #]
 for all [x: Int]
 $x < (x + 1)$
 implies Rational without [$^{-1}$, /] with [Int for R]
 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 \odot , - # for inv, - for \oslash , R for T],
 InfixInverse with [* for \odot , # $^{-1}$ for inv, / for \oslash , 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, *, # - #, /, =, \leq , \geq , <, >]

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 $>$]

Enumerated Data Types**Enumerated: trait****imports** Ordinal**includes** Ordered**introduces**first: $\rightarrow T$ last: $\rightarrow T$ succ: $T \rightarrow T$ pred: $T \rightarrow T$ ord: $T \rightarrow \text{Ord}$ **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

 $x \leq y = \text{ord}(x) \leq \text{ord}(y)$ **implies** T generated by [last, pred]

for all [x: T]

succ(pred(x)) = if x = first then succ(first) else x

first $\leq x$ $x \leq \text{last}$ **converts** [=, \leq , \geq , $<$, $>$]**Rainbow: trait****includes** Enumerated with [Color for T]**introduces**red: $\rightarrow \text{Color}$ orange: $\rightarrow \text{Color}$ yellow: $\rightarrow \text{Color}$ green: $\rightarrow \text{Color}$ blue: $\rightarrow \text{Color}$ violet: $\rightarrow \text{Color}$ **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, =, \leq , \geq , $<$, $>$, red, orange, yellow, green, blue, violet],[succ, first, ord, =, \leq , \geq , $<$, $>$, red, orange, yellow, green, blue, violet]**Character: trait** includes Enumerated with [Char for T]

% For each programming language there will be mappings from character and string constants to
 % terms in the shared language. Because of the variety of character orderings and notations for
 % constants, these definitions are not likely to be portable across programming languages.

Display Traits

% The following traits represent a fairly straightforward translation of the specifications in
 % "Formal Specification as a Design Tool" (CSL-80-1). 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, cdl: Coordinate ]
      appearance(addComponent(cn, cm, cdl), cd) =
        if in(cm, minus(cd, cdl))
          then (if in(cn, cd)
            then combine(appearance(cm, minus(cd, cdl)),
              appearance(cn, cd))
            else appearance(cm, minus(cd, cdl)))
          else appearance(cn, cd)
      ~in(empty, cd)
      in(addComponent(cn, cm, cdl), cd) =
        in(cm, minus(cd, cdl)) | 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, coerceView for coerce],
 ComponentCoercion with [Text for T, coerceText for coerce],
 ComponentCoercion with [Figure for T, coerceFigure for coerce]

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, cdl: Coordinate, id, idl: PictureId, p: Picture]
 appearance(addPicture(v, cdl, id, p), cd) =
 if in(p, minus(cd, cdl)) then appearance(p, minus(cd, cdl))
 else appearance(v, cd)
 ~in(empty, cd)
 in(addPicture(v, cdl, id, p), cd) = (in(p, minus(cd, cdl)) | in(v, cd))
 findPictures(empty, cd) = new
 findPictures(addPicture(v, cdl, id, p), cd) =
 if in(p, minus(cd, cdl)) then insert(id, findPictures(v, cd))
 else findPictures(v, cd)
 deletePicture(empty, id) = empty
 deletePicture(addPicture(v, cdl, idl, p), id) =
 if id .eq idl then v else addPicture(deletePicture(v, id), cd, idl, 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

References

[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, Englewood Cliffs, 1978.

[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*, Cambridge, MA, 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.

[Forgaard 83]

R. Forgaard, "A Program for Generating and Analyzing Term Rewriting Systems," S.M. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, to appear, 1983.

[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, 1-149.

[Guttag and Horning 80]

J.V. Guttag and J.J. Horning, "Formal Specification as a Design Tool," *Proc. ACM Symposium on Principles of Programming Languages*, Las Vegas, Jan. 1980, 251-261.

[Guttag and Horning 83]

J.V. Guttag and J.J. Horning, "An Introduction to the Larch Shared Language," *Proc. IFIP Congress '83*, Paris, 1983.

[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.

[Kownacki 83]

R. Kownacki, "A User's Guide to the Larch Shared Language Specification Checker," Laboratory for Computer Science, Massachusetts Institute of Technology, to appear, 1983.

[Lescanne 83]

P. Lescanne, "Computer Experiments with the REVE Term Rewriting System Generator," *Proc. ACM Symposium on Principles of Programming Languages*, Austin, Jan. 1983, 99-108.

[Musser 80]

D.R. Musser, "Abstract Data Type Specification in the Affirm System,"
IEEE Transactions on Software Engineering vol. 1, 1980, 24-32.

[Wand 79]

M. Wand, "Final Algebra Semantics and Data Type Extensions,"
Journal of Computer and System Sciences, vol. 19, 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.

[Zachary 83]

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