8-3-2001

Larch Frequently Asked Questions

Gary T. Leavens

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons, Software Engineering Commons, and the Theory and Algorithms Commons

Recommended Citation

http://lib.dr.iastate.edu/cs_techreports/284

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
Larch Frequently Asked Questions

by Gary T. Leavens
Introduction

This document is a list of frequently asked questions (FAQ) and their answers for the Larch family of specification languages. It is intended to be useful to those who are new to Larch, especially to students and others trying to understand and apply Larch. However, there is some material that is also intended for experts in other formal methods who would like to learn more about Larch. If you find something that seems too technical, please skip a bit, perhaps to another question.

We are looking for more contributions of questions and answers, as well as corrections and additions to the answers given. Please send e-mail to us at larch-faq@cs-DOT-iastate-DOT-edu (after changing each “.DOT.” to “.”). We will welcome and acknowledge any help you may give.

Bibliographic references are described at the end of this document (see [Bibliography], page 65). They look like “[Wing83]”, which give the names of up to 3 authors, and an abbreviated year of publication.

This FAQ is accessible on the WWW in the following URL.

http://www.cs.iastate.edu/~leavens/larch-faq.html

HTML, plain text, postscript, and GNU info format versions are also available by anonymous ftp at the following URLs.


(In the postscript version of this manual, the table of contents is printed at the end. If the table of contents has not already been moved for you, please take those pages out and put them right after the title and copyright pages in the front.)
Acknowledgements

This work was partially funded by (US) National Science Foundation, under grant CCR-9503168.

Thanks to Matt Markland who provided the initial impetus for getting this FAQ started, who helped make up the initial list of questions.

Many thanks to John Guttag, Jim Horning, Jeannette Wing, and Steve Garland for creating the Larch approach and its tools, and for their patient teaching of its ideas through their writing, personal contact, and many e-mail messages.

Thanks to Perry Alexander, Al Baker, Pieter Bekaert, Eric Eide, John Fitzgerald, Jim Horning, Ursula Martin, Bertrand Meyer, Clyde Ruby, and Jeannette Wing for general comments, updates, and corrections to this FAQ. Thanks to Steve Garland and Jim Horning for help with answers. Thanks to Pierre Lescanne, Kishore Dhara, Teo Rus, Narayanan Sridhar, and Sathiyanarayanan Vijayaraghavan for suggesting questions.
1 Larch and the Larch Family of Languages

In this chapter, we discuss global questions about Larch.

1.1 What is Larch? What is the Larch family of specification languages?

Larch [Guttag-Horning93] may be thought of as an approach to formal specification of program modules. This approach is an extension of Hoare’s ideas for program specification [Hoare69] [Hoare72a]. Its distinguishing feature is that it uses two “tiers” (or layers) [Wing83]. The top tier is a behavioral interface specification language (BISL), tailored to a specific programming language. Such BISLs typically use pre- and postcondition specifications to specify software modules. The bottom tier is the Larch Shared Language (LSL, see [Guttag-Horning93], Chapter 4), which is used to describe the mathematical vocabulary used in the pre- and postcondition specifications. The idea is LSL specifies a domain model, some mathematical vocabulary, and the BISL specifies both the interface and the behavior of program modules. See Section 1.2 [Why does Larch have two tiers?], page 4, for more of the reasons for this separation.

The Larch family of specification languages [Guttag-Horning-Wing85b] consists of all the BISLs that use LSL for specification of mathematical vocabulary. Published BISLs with easily accessible references include:

- Larch/CLU [Wing83] [Wing87], for CLU,
- Larch/Ada [Guaspari-Marceau-Polak90] [ORA94], for Ada,
- LCL [Guttag-Horning93] Chapter 5 and [Tan94] [Evans00], for ANSI C,
- LM3 [Guttag-Horning93] Chapter 6 and [Jones91] [Jones93], for Modula-3,
- Larch/Smalltalk [Cheon-Leavens94] [Cheon-Leavens94b], for Smalltalk-80,
- Larch/C++ [Leavens97] [Leavens96b], for C++, and
- Larch/ML [Wing-Rollins-Zaremski93], for Standard ML,
- VSPEC [Baraona-Penix-Alexander95] [Baraona-Alexander96] [Baraona-Alexander-Wilsey96], for VHDL.

There are also some BISLs whose documentation is not as easy to come by: Larch/Generic [Chen89], GCIL (which stands for Generic Concurrent Interface specification Language [Lerner91]), and Larch/CORBA (a BISL for CORBA IDL [Sivaprasad95]).

Typically, each BISL uses the declaration syntax of the programming language for which it is tailored (e.g., C for LCL, Modula-3 for LM3), and adds annotations to specify behavior. These annotations typically consist of pre- and postconditions (often using the keywords requires and ensures), some way to state a frame axiom (often using the keyword modifies), and some way to indicate what LSL traits are used to provide the mathematical vocabulary. See Chapter 4 [Behavioral Interface Specification Languages (BISLs)], page 52, for more details.
1.2 Why does Larch have two tiers?

The two “tiers” used in the Larch family of specification languages are LSL, which is called the bottom tier, and a behavioral interface specification language (a BISL), which is called the top tier.

This specification of program modules using two “tiers” is a deliberate design decision (see [Wing83] and [Guttag-Horning93], Chapter 3). It is similar to the way that VDM [Jones90] [ISO-VDM96] [Fitzgerald-Larsen98] is typically used (one specifies a model and some auxiliary functions, in a way similar to LSL, and then one uses that vocabulary to specify the operations that are to be implemented in some programming language.) However, VDM does not have a family of BISLs that are tailored to specific programming languages. Only the Larch family, and the RESOLVE family [Edwards-etal94], have specification languages tailored to specific programming languages. (See Section 1.4 [How does Larch compare to other specification languages?], page 6, for more comparisons.)

What are the advantages of using the Larch two-tiered approach?

- Having different BISLs tailored to each programming language allows each BISL to specify all the details of a program module’s interface (how to call it) and behavior (what it does). If one has a generic interface specification language, such as VDM-SL [Jones90] [ISO-VDM96] [Fitzgerald-Larsen98], then one cannot specify all interface details.

- The division into two tiers allows the language used for each tier to be more carefully designed and expressive. For example, although it is possible to deal with finiteness, partiality, exceptional behavior, aliasing, mutation, and side-effects in mathematics, dealing with such issues requires using more apparatus than just mathematical functions. (One might use, for example, the techniques of denotational semantics [Schmidt86].) Since such additional apparatus does not fit well with an algebraic style of specification, these issues are thus typically ignored in LSL specifications. By ignoring these issues, LSL specifications can deal with potentially infinite, pure values, and total functions, in a stateless setting. On the other hand, while pre- and postcondition specifications are good for the specification of finiteness, partiality, exceptions, aliasing, mutation, and side-effects, they are not well adapted to the specification of mathematical vocabulary. Thus each tier uses techniques that are most appropriate to its particular task.

- The division into two tiers allows the underlying logic to be classical. That is, because one is not specifying recursive programs in LSL, there is no need for a specialized logic that deals with partial functions; instead, LSL can use total functions and classical logic [Leavens-Wing97].

- The domain models described in LSL can be shared among many different BISLs, so the effort that goes into constructing such models is not limited to just one BISL [Wing83].

One could imagine using some other language besides LSL for the bottom tier and still getting these advantages. This is done, for example, in RESOLVE [Edwards-etal94]. Chalin [Chalin95] has suggested that Z [Hayes93] [Spivey92] could also serve for the bottom tier, but one would probably want to use a subset of Z without state and the specification of side-effects for that.
The major disadvantage of the tailoring of each BISL to a specific programming language is that each BISL has its own syntax and semantics, and thus requires its own tool support, manuals, etc.

(Parts of the following paragraph are adapted from a personal communication from Horning of November 29, 1996.)

Something that might seem to be a disadvantage of the Larch approach is that one sometimes finds oneself writing out a very similar specifications in both LSL and a BISL. Isn’t that a waste of effort? It may seem like that, but one has to realize that one is doing two different things, and thus the two similar specifications cannot really be the same. What may happen is that one writes similar declarations on the two levels, with parallel sets of operators and procedures. For example, one might specify LSL operators `empty`, `push`, `pop`, `top` for stacks, and then specify in a BISL procedures `Empty`, `Push`, `Pop`, and `Top`. While the BISL procedure specification will use the LSL operators in their pre- and postconditions, there will probably be significant differences in their signatures. For example, the procedure `Push` may have a side-effect, rather than returning a new stack value as would the `push` operator in the trait. If you find yourself repeating the axioms from an LSL trait in a BISL specification, then you’re probably making a mistake.

It is also difficult to extract a simple mathematical vocabulary (like the total functions of LSL) from a BISL specification. This is because the semantics of program modules tends to be complex. Thus a BISL specification is not an appropriate vehicle for the specification of mathematical vocabulary.

See Section 1.3 [What is the difference between LSL and a Larch BISL?], page 5, for more discussion on this point.

1.3 What is the difference between LSL and a Larch BISL?

The main difference between LSL and a Larch BISL is that in LSL one specifies mathematical theories of the operators that are used in the pre- and postcondition specifications of a Larch BISL. Thus LSL is used to specify mathematical models and auxiliary functions, and the a Larch BISL is used to specify program modules that are to be implemented in some particular programming language.

The following summary, provided by Horning (private communication, October 1995), contrasts LSL and a Larch BISL.

<table>
<thead>
<tr>
<th>A BISL specifies (not LSL)</th>
<th>LSL specifies (not a BISL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Theory definition</td>
</tr>
<tr>
<td>storage allocation</td>
<td>Theory operations</td>
</tr>
<tr>
<td>aliasing, side-effects</td>
<td>inclusion, assumption,</td>
</tr>
<tr>
<td>Control</td>
<td>implication, parameterization</td>
</tr>
<tr>
<td>errors, exception handling,</td>
<td>Operator definitions</td>
</tr>
<tr>
<td>partiality, pre-conditions,</td>
<td>Facilities for libraries, reuse</td>
</tr>
<tr>
<td>concurrency</td>
<td></td>
</tr>
<tr>
<td>Programming language constructs</td>
<td></td>
</tr>
<tr>
<td>type system, parameter modes,</td>
<td></td>
</tr>
<tr>
<td>special syntax and notations</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
</tr>
</tbody>
</table>
Horning’s summary is that a BISL handles “all the messy, boring stuff”, while LSL handles “all the subtle, hard stuff”, and thus the features in the two tiers do not overlap much. However, that summary is from the point-of-view of a designer of LSL. From the point of view of a user of a Larch BISL, the BISL is where you specify what is to be implemented, and LSL is where you do “domain modeling”.

See [Lamport89] [Guttag-Horning93], Chapter 3, for more about the role of interface specification, and the separation into two tiers.

1.4 How does Larch compare to other specification languages?

First, a more precise comparison is needed, because Larch is not a single language, but a family of languages (see above). Another problem with this comparison is that Larch has two tiers, but VDM-SL [Jones90] [ISO-VDM96] [Fitzgerald-Larsen98], Z [Hayes93] [Spivey92], and COLD-K [Feijs-Jonkers92] are all integrated languages, which mix aspects of both of the tiers of the Larch family. Thus we will compare some aspects of VDM-SL and Z to LSL and other aspects to BISLs in the Larch family (such as LCL or LM3 [Guttag-Horning93], Chapters 5 and 6).

(The comparisons below tend to be a bit technical; if you are not familiar with specification and verification, you might want to skip to another question.)

An excellent resource for comparison of LSL to other algebraic specification languages is [Mosses96]. Also a comparison of Larch with RESOLVE [Edwards-etal94] is found in [Sitaraman-Welch-Harms93].

1.4.1 How does Larch compare to VDM-SL?

By VDM, one means, of course, the specification language VDM-SL [Jones90] [ISO-VDM96] [Fitzgerald-Larsen98]. In comparison with LSL, in VDM-SL one can specify mathematical values (models) using constructs similar to those in denotational semantics and typed, functional programming languages. That is, one can declare types of records, tagged unions, sets, maps and so on, and one can specify mathematical functions on such values, either in a functional style, or by the use of pre- and postcondition specifications. For example, one might write the following in VDM-SL to define a phone book.

```
-- This is a VDM-SL model and auxiliary function specification

PhoneBook = map Id to (Info | HIDDEN)
Info :: number : seq of Digit
      address : Address

listed : PhoneBook * Id -> bool
listed(pb, id) == id in set dom pb
```

The above example uses the interchange syntax of VDM-SL [ISO-VDM96] is used. In this syntax, for example, * is an approximation to the mathematical notation for a cross product; see Section 10 of [ISO-VDM96] for details. The type PhoneBook is a map to a
tagged union type. The type \texttt{Info} is a record type. The auxiliary function \texttt{listed} is given in a mathematical form, similar to LSL.

In LSL one can use standard traits (for example, those in Guttag and Horning’s handbook [Guttag-Horning93], Appendix A) and the LSL shorthands \texttt{tuple} and \texttt{union} (see Section 4.8 of [Guttag-Horning93]) to translate VDM-SL specifications of mathematical values into LSL. For example, the following is an approximation to the above VDM-SL specification.

\begin{verbatim}
% This is LSL

PhoneBookTrait : trait
  includes Sequence(Digit, Number),
    FiniteMap(PhoneBook, Id, InfoOrHIDDEN)
Info tuple of number : Number, address : Address
InfoOrHidden union of left : Info, right : HIDDEN
introduces
  listed : PhoneBook, Id -> Bool
asserts
  \forall pb: PhoneBook, id: Id
  listed(pb, id) = defined(pb, id);

However, VDM-SL does not have the equivalent of LSL’s equational specification constructs, and thus it is more difficult in VDM-SL to specify a mathematical vocabulary at the same level of abstraction. That is, there is no provision for algebraic specification in VDM-SL.

Because VDM-SL has a component that is like a programming language, and because it allows recursive definitions, the logic used in VDM-SL is a three-valued logic instead of the classical, two-valued logic of LSL. This may make reasoning about VDM-SL specifications relatively more difficult than reasoning about LSL specifications.

In comparison to Larch family BISLs, the first thing to note is that VDM-SL is not a BISL itself, as it is not tailored to the specification of interfaces for some particular programming language. This is evident in the following example, which continues the above VDM-SL phonebook specification.

\begin{verbatim}
-- The following is a VDM-SL operation specification

LOOKUP(pb: PhoneBook, id: Id) i: Info
pre listed(pb, id)
post i = pb(id)
\end{verbatim}

In the above VDM-SL specification, it is evident that one can define some generic interface information (parameters, information about their abstract values, etc.). The pre- and postcondition format of such specifications has been adopted by most Larch family BISLs. Framing for procedure specifications in VDM-SL is done by declaring external variables, and noting which are readable and writable by a procedure; these two classes of variables are respectively defined by external declarations (in LCL and Larch/C++), and by the \texttt{modifies} clause. Because VDM-SL is generic, it cannot, by itself, specify language-specific interface details, such as pointers, exception handling, etc. However, an advantage of VDM-SL is that it has better tool support than most BISLs in the Larch family.
1.4.2 How does Larch compare to Z?

Like VDM-SL, Z [Hayes93] [Spivey92] (pronounced “zed”) is a specification language that allows both the specification of mathematical values and program modules.

Like LSL, Z allows the definition of mathematical operators equationally (see [Spivey92], section 3.2.2). A Z schema is roughly comparable to an LSL trait. The main difference between Z and LSL is that in Z specifications can include state variables. That is, a Z schema can specify variables, or a procedure with side effects on such variables. In this respect a Z schema can act much like a Larch family BISL, since, unlike LSL, it is not restricted to the specification of mathematical constants and mathematical functions.

By concentrating on the features of Z that do not involve state variables, one can more closely compare Z and LSL. We do this for the next several paragraphs below.

The Z schema calculus (see [Spivey92], section 3.8) allows one to apply to schemas: most logical connectives, quantification, name hiding, and pipelining. (One can also use sequential composition between schemas that specify side-effects.) The schema calculus is thus more expressive than the LSL mechanisms: includes and assumes. However, in LSL one can effectively conjoin traits by including them all to make a new trait.

It is possible in Z to write schemas that describe constants and mathematical functions only. However, many purely mathematical Z schemas describe something more like an instance of an LSL tuple. This is done in Z by giving several variable declarations. For example, consider the following Z schema (in which Z stands for the integers).

```plaintext
--Z_rational----------------------------------
| num, den: Z
---------------------------------------------
```

The above is semantically similar to the LSL trait LSL_rational given below.

```plaintext
LSL_rational: trait
    includes Integer
    introduces
    num: -> Int
    den: -> Int
```

However, in Z usage, the schema Z_rational is likely to be useful, whereas in LSL one would typically write a trait such as the following, which specifies a type (called a sort in LSL) of rational numbers.

```plaintext
LSL_rational_sort: trait
    includes Integer
    rational tuple of num: Int, den: Int
```

The above is semantically similar to a Z schema that defines rational as the Cartesian product of Z and Z, and specifies the constructors and observers of the LSL shorthand (see [Guttag-Horning93], Figure 4.10). This is a difference in usage, not in expressiveness, however.

Z and LSL both have comparable built-in notations for describing abstract values. Z has basic types (the integers, booleans, etc.) and more type constructors (sets, Cartesian products, schemas, and free types). A “free type” in Z (see [Spivey92], Section 3.10) is similar to an LSL union type, but Z allows direct recursion. Z also has a mathematical tool-kit (see [Spivey92], Chapter 4) which specifies: relations, partial and total functions, sequences,
and bags, and a large number of operation symbols and mathematical functions defined on these types. This tool-kit is roughly comparable to Guttag and Horning's handbook (see [Guttag-Horning93], Appendix A), although it tends to make heavier use of symbols (like “*”) instead of named functions (like “plus”). (This use of symbols in Z has been found to be a problem with the understandability of Z specifications [Finney96]. However, the readability of Z is improved by the standard conventions for mixing informal commentary with the formal notation.)

LSL and Z both use classical logic, and the underspecification approach to the specification of partiality, in contrast to VDM-SL. Z does not have any way of stating the equivalent of an LSL partitioned by or generated by clause.

See [Baumeister96] for an extended Z example (the birthday book of [Spivey92]) worked in LSL.

Compared to a Larch family BISL, Z, like VDM-SL is generic, and thus is not able to specify the interface details of a particular programming language.

An interesting semantic difference between Z and a typical Larch-family BISL (and VDM-SL) is that Z does not have true preconditions (see [Jackson95], Sections 7.2 and 8.1); instead, Z has guards (enabling conditions, which are somewhat like the when clauses of GCIL [Lerner91]). Thus Z may be a better match to the task of defining finite state machines; the use of enabling conditions also (as Jackson points out) allows different views of specifications to be combined. Aside from the Larch BISLs like GCIL that have support for concurrency, most Larch family BISLs have preconditions which are “disclaimers”. Because of this difference, and the Z schema calculus, it is much easier to combine Z specifications than it is to combine specifications in a Larch BISL.

A related point of usage difference is the frequency of the use of invariants in Z (Spivey calls these “constraints”; see [Spivey92], section 3.2.2). Because in Z one often specifies schemas with state variables, it makes sense to constrain such variables in various ways. In Z, a common idiom is to conjoin two schemas, in which case the invariants of each apply in the conjoined schema. A typical Larch family BISL has no provision for such conjunction, although object-oriented BISLs (such as Larch/C++) do get some mileage out of conjunction when subtypes are specified.

Because Z does not distinguish operation specifications from auxiliary function specifications, it does not have a separation into tiers. So in Z one can use operations previously specified to specify other operations.

1.4.3 How does Larch compare to COLD-K?

Like Larch, COLD-K [Feijs-Jonkers92] makes more of a separation into mathematical and interface specifications, although all are part of the same language. The part of COLD-K comparable to LSL is its algebraic specifications (see [Feijs-Jonkers92], Chapters 2 and 3). In contrast to LSL, COLD-K does not use classical logic, and can specify partial functions. All COLD-K types have an “undefined” element, except the type of the Booleans. The logic in COLD-K has a definedness predicate that allows one to specify that some term must be undefined, for example; such a specification is not directly supported by LSL. A feature of COLD-K not found in LSL is the ability to hide names (see [Feijs-Jonkers92], Section 3.3); that is to only export certain names from a COLD-K scheme. Both COLD-K and LSL have mechanisms for importing and renaming.
Compared to a Larch family BISL, COLD-K is often more expressive, because it uses dynamic logic as its fundamental semantics (see [Feijs-Jonkers92], Section 5.9). In COLD-K one can also write algorithmic descriptions, and can mix algebraic, algorithmic, and dynamic logic specifications. COLD-K has relatively more sophisticated mechanisms for framing (see [Feijs-Jonkers92], p. 126, and [Jonkers91]) than most Larch family BISLs (see Section 4.9 [What does modifies mean?], page 57). All of this would seem to make COLD-K both more expressive and difficult to learn than a typical Larch family BISL.

1.5 What is the history of the Larch project?

(The following is adapted from a posting of Horning to the larch-interest mailing list on June 19, 1995, which was itself condensed from the preface of [Guttag-Horning93].)

This project has been a long time in the growing. The seed was planted by Steve Zilles on October 3, 1973. During a programming language workshop organized by Barbara Liskov, he presented three simple equations relating operations on sets, and argued that anything that could reasonably be called a set would satisfy these axioms, and that anything that satisfied these axioms could reasonably be called a set. John Guttag refined this idea. In his 1975 Ph.D. thesis (University of Toronto), he showed that all computable functions over an abstract data type could be defined algebraically using equations of a simple form. He also considered the question of when such a specification constituted an adequate definition.

As early as 1974, Guttag and Horning realized that a purely algebraic approach to specification was unlikely to be practical. At that time, they proposed a combination of algebraic and operational specifications which they referred to as “dyadic specification.” Working with Wing, by 1980 they had evolved the essence of the two-tiered approach to specification; the term "two-tiered" was introduced by Wing in her dissertation [Wing83]. A description of an early version of the Larch Shared Language was published in 1983. The first reasonably comprehensive description of Larch was published in 1985 [Guttag-Horning-Wing85a].

In the spring of 1990, software tools supporting Larch were becoming available, and people began using them to check and reason about specifications. To make information on Larch more widely available, it was decided to write a book [Guttag-Horning93].

The First International Workshop on Larch, organized by Ursula Martin and Jeannette Wing, was held in Dedham, Massachusetts, July 13-15, 1992 [Martin-Wing93].

1.6 What is the origin of the name Larch?

According to Jim Horning (personal communication, January 20, 1998) and John Guttag (personal communication, March 28 1998): “The name was not selected at PARC (hence from the Sunset Western Garden Book), but at MIT. The project had been known informally there as ‘Bicycle’.” One day Mike Dertouzos [director of the MIT Laboratory for Computer Science] and John were talking on the phone. According to Jim, Mike asked John to “think up a new name quick, because he could just imagine what Senator Proxmire would say if he noticed that DARPA was sponsoring ‘a bicycle project’ at MIT.” John was, he says, “looking at a larch tree at” his “parent’s house” and came up with “Larch”. He also says “I did grow up in Larchmont, and I’m sure that influenced my choice of name.” Jim adds,
“It was only later that Butler Lampson suggested that we could use it for the specification of very ‘larch’ programs.”

The “Larch” is the common name of a species of fir tree. The cover of [Guttag-Horning93] has a picture of Larch trees on it. For more pictures of Larch trees, see the following URL.

http://www.cs.iastate.edu/~leavens/larch-pics.html

GIF format pictures of the Larch logo, can be found at the following URLs.

http://www.sds.lcs.mit.edu/spd/larch/pictures/larch.gif
http://www.sds.lcs.mit.edu/spd/larch/pictures/trees.gif

1.7 Where can I get more information on Larch and Larch languages?

A good place to start is the Guttag and Horning’s book [Guttag-Horning93]. (This book is sometimes called “the silver book” by Larchers.) Consider it required reading. If you find that all tough going, you might want to start with Liskov and Guttag’s book [Liskov-Guttag86], which explains the background and motivates the ideas behind abstraction and specification. (See Section 1.9 [What is the use of formal specification and formal methods?], page 12, for more background.)

You might also want to read the introductory article about the family [Guttag-Horning-Wing85b], although it is now somewhat dated. You might (especially if you are an academic) want to also read some of the proceedings of the First International Workshop on Larch [Martin-Wing93], which contains several papers about Larch.

The usenet newsgroup ‘comp.specification.larch’ is devoted to Larch. You might also want to read the newsgroup ‘comp.specification.misc’ for more general discussions.

There was a mailing list, called “larch-interest,” for Larch. However, it has been discontinued, now that the usenet newsgroup is available. The archives of this list are available from the following URL.


Other on-line resources can be found through the global home page for Larch at the following URL.

http://www.sds.lcs.mit.edu/spd/larch/index.html

Other resources are SRC’s Larch home page


and Horning’s page.

http://www.star-lab.com/horning/larch.html

This frequently asked questions list (FAQ) is accessible on the world-wide web in:

http://www.cs.iastate.edu/~leavens/larch-faq.html

HTML, plain text, postscript, and GNU info format versions are also available by anonymous ftp at the following URLs.

1.8 Is there an object-oriented extension to Larch?

This question might mean one of several other more precise questions. These questions are discussed below.

One question is: is there a Larch-style BISL for some particular object-oriented (OO) programming language? Yes, there are Larch-style BISLs for Modula-3 (see [Guttag-Horning93], chapter 6, and [Jones91]), Smalltalk-80 (see [Cheon-Leavens94]), and C++ (see [Leavens97] [Leavens96b]).

Another question is: does LSL have extensions that support object-oriented specification? No, there are none in particular. One could imagine extensions of LSL with some notion of subsorting, as in OBJ [Futatsugi-etal85] [Goguen-etal87], but as yet this has not been done.

1.9 What is the use of formal specification and formal methods?

There are endless debates about the use of formal specification and formal methods. For recent discussions see [Saiedian-etal96]. The following discussion may help you enter the debate in a reasonable fashion.

First, be sure you know what formal methods are, and what they are good for [Liskov-Guttag86] [Wing90]. Roughly speaking, formal methods encompass formalized software design processes, formal specifications, formal derivation of implementations, and formal verification. You might also want to look at the following URL for more recent work on formal methods.

http://www.comlab.ox.ac.uk/archive/formal-methods.html

Formal verification of implementations that were not designed with verification in mind is impossible in general. Formal verification has been critiqued as too difficult to justify and manage [DeMillo-Lipton79]. Programs can also only be verified relative to some abstract machine [Fetzer88]. However, these problems haven’t stopped people from trying to use formal verification to gain an added level of confidence (or to help find bugs [Garland-Guttag-Horning90] [Tan94]) in software that is very critical (in terms of safety or business factors). Of course, you can still use formal methods without doing formal verification.

Several advocates of formal methods have tried to explain formal methods by debunking the “myths” that surround them [Hall90] [Bowen-Hinchey95]. Bowen and Hinchey have also given some advice in the form of “commandments” for good use of formal methods [Bowen-Hinchey95b].

Of the various kinds of formal methods, formal specification (that’s where Larch comes in), seem to be the most useful. At a fundamental level, some specification is needed for any abstraction (otherwise there is no abstraction [Liskov-Guttag86]). A specification is useful as a contract between the implementor and a client of an abstraction (such as a procedure or a data type). This contract lays out the responsibilities and obligations of both parties, and allows division of labor (and a division of blame when something goes wrong). However, such a specification need not be formal; the main reason for preferring formal specification over informal specification is that formal specification can help prevent the ambiguity and unintentional imprecision which plague informal specification.
There are, however, some disadvantages to using formal specifications. One shouldn’t take the search for precision to an extreme and try to formalize *everything*, as this will probably cost much time and effort. (Instead, try to formalize just the critical aspects of the system.) Another disadvantage is that a formal specification can itself contain errors and inconsistencies, which can lead to the specification becoming meaningless. The use of formality by itself also does not lead to well-structured and readable specifications; that is a job that humans have to do.

Although you can use the Larch languages and tools without subscribing to any particular view on the utility of formal methods, there is an emerging view in the Larch community. This view is that formal specification and formal methods are most useful as aids to debugging designs [Garland-Guttag-Horning90] [Tan94]. The basic idea is that one writes (what one thinks is) redundant information into specifications, and this information is compared with the rest of the specification. Often the supposedly redundant information does not match, and trying to check that it does brings out design or conceptual errors. Thus formal methods play a role complementary to prototyping, in that they allow one to analyze certain properties of a design to see if there are lurking inconsistencies or problems.

Much of what has been said above seems logical, but experimentation would be helpful to understand what is really true and to help quantify the costs and benefits of formal methods.

One last answer to the question: some people find writing formal specifications (and even proofs) fun. No one knows why.
2 The Larch Shared Language (LSL)

In this chapter, we discuss questions about the Larch Shared Language (LSL).

2.1 What is the Larch Shared Language (LSL)?

The Larch Shared Language (LSL) (see [Guttag-Horning93], Chapter 4, and [Guttag-Horning-Modet90]) is a language for specifying mathematical theories. LSL is a kind of equational algebraic specification language [Guttag75] [Guttag-Horning78] [Goguen-Thatcher-Wagner78] [Ehrig-Mahr85] [Futatsugi-etal85] [Mosses96] [Loeckx-Ehrich-Wolf96]. That is, specifications in LSL mainly consist of first-order equations between terms. The semantics of LSL is based on classical, multisorted first-order equational logic (see Section 2.14 [What is the meaning of an LSL specification?], page 23). However, LSL does have two second-order constructs, which allow you to specify rules of data type induction and certain kinds of deduction rules (see Section 2.10 [How and when should I use a generated by clause?], page 19, and see Section 2.9 [How and when should I use a partitioned by clause?], page 18).

Unlike a programming language, LSL is a purely declarative, mathematical formalism. In LSL there are no side-effects, algorithms, etc. (see Section 2.18 [What pitfalls are there for LSL specifiers?], page 27). Instead of specifying computation, in LSL one specifies mathematical operators and their theories. These operators are used in Larch behavioral interface specification languages as their mathematical vocabulary (see Section 1.1 [What is Larch?], page 3).

An unusual feature of LSL is that it provides ways to add checkable redundancy to specifications (see Section 2.11 [When should I use an implies section?], page 20). See Section 1.4 [How does Larch compare to other specification languages?], page 6, for more detailed comparisons of LSL and related specification languages.

See Chapter 4 of [Guttag-Horning93], and the rest of this chapter, for more information.

2.2 Where can I find information on-line about LSL?

Besides this FAQ, the best place to look is probably your own computer system. You should have a manual page for the LSL checker, if it’s installed on your system. Try the Unix command `man lsl` to see it.

You should also look for a directory (such as `/usr/larch/LSL`) where the installation of LSL is found. In that directory, you will find a subdirectory ‘Doc’, where there is some documentation on the checker. See Section 2.3 [Where can I get a checker for LSL?], page 15, if you don’t have the LSL checker installed on your system.

There is no official reference manual for the latest version of LSL (3.1) on-line. The defining document for LSL version 2.3 [Guttag-Horning-Modet90] is available from the URL

2.3 Where can I get a checker for LSL?

You can get the MIT releases of the LSL checker using the world-wide web, starting at
the following URL.

http://www.sds.lcs.mit.edu/spd/larch/lsl.html

You can also get the MIT release by anonymous ftp from the following URL.

ftp://ftp.sds.lcs.mit.edu/pub/Larch/LSL/

At Iowa State, the Larch/C++ group has made available later beta releases of the LSL
checker that fix problems with its profligate use of space. You can get the sources for Unix
and Windows 95 executables from the following URL.


You might also be interested in the Larch Development Environment (LDE), from
Cheng’s group at Michigan State University. This environment integrates tools that sup-
port LSL, LP, LCL, and Larch/C++. It runs under SunOS and Solaris. You can get it from
the following URL.


2.4 Do you have a good makefile to use with the LSL
checker?

The following makefile shows one way to use the Unix command make to help check LSL
traits. It relies on properties of the Bourne shell under Unix and standard Unix make, so
it would have to be adjusted to work on other systems. (Also, if you cut and paste this
makefile, be sure to change the leading blanks to tab characters on the rule lines.)

SHELL = /bin/sh
LSL = lsl
LSLFLAGS =

.SUFFIXES: .lsl .lsl-ckd .lp
.lsl.lsl-ckd:
   $(LSL) $(LSLFLAGS) $< 2>&1 | tee .lsl.lp:
   $(LSL) -lp $(LSLFLAGS) $<

checkalltraits:
   $(LSL) $(LSLFLAGS) *.lsl

clean:
   rm -f *.lsl-ckd *.lpfrz
cleanall: clean
   rm -f *.lpscr *.lp *.lplog

If you use the emacs text editor, you can then check your traits by using M-x compile
(and responding with something like checkalltraits and a return). When you do this,
you can then use C-x ‘ (which is an alias for M-x next-error) to edit the LSL trait with
the next error message (if you’re using a version of the LSL checker version 3.1beta5 or
later). Emacs will put the cursor on the appropriate line for you automatically.

See Section 3.10 [How do I use LP to check my LSL traits?], page 43, for more information
on how to use LP to check the implications in LSL traits.
2.5 What is a sort?

The “sorts” in an LSL trait correspond to the types in a programming language. They are names for sets of values.

This usage is historical (it seems to come from [Goguen-Thatcher-Wagner78]), and arises from the desire to avoid the word “type”, which is heavily overused. However, it does have some practical benefit, in that in the context of a BISL, one can say that a type comes from a programming language, but a sort comes from LSL.

Unlike some other specification languages (e.g., OBJ [Futatsugi-etal85] [Goguen-etal87]), in LSL there is no requirement that all sort names be declared. For example, in the following trait, a sort Answer is used.

```
AnswerTrait: trait
  introduces
  yes, no, maybe: -> Answer
```

In summary, LSL sorts are declared by being mentioned in the signatures of operators.

2.6 What is the purpose of an LSL trait? What is a trait used for?

An LSL trait is used to describe a mathematical theory. This could be used by a mathematician to simply formalize a theory, but more commonly the theory specified is intended to be used as the mathematical vocabulary for some BISL. Another common use is as a way of specifying input to the Larch Prover (LP).

When used as mathematical vocabulary for some behavioral interface specification, one can identify some other common uses. Quite often one wants to specify the abstract values of some data type. (See Section 4.4 [What is an abstract value?], page 55.) Examples include the Integer, Set, Queue, and String traits in Guttag and Horning’s handbook [Guttag-Horning93a]. Another common use is to specify some function on abstract values, in order to ease the specification of some procedure.

One might also write a trait in order to:

- specialize some existing trait by renaming some sorts or by instantiating some parameterized sorts,
- change vocabulary by renaming some operators of some existing trait,
- state implications, for the benefit of readers, to help debug a specification [Garland-Guttag-Horning90], or as input to LP,
- add additional operators to some existing trait, as a convenience in specifying some interface or as a way of introducing a name for some idea that is an important concept in an interface specification, or
- combine several existing traits, to make it easier to use them in an interface specification.
2.7 What are the sections of an LSL trait?

The sections of an LSL trait are determined by the LSL grammar [Guttag-Horning-Modet90]. (See Section 2.3 [Where can I get a checker for LSL?], page 15, for a more recent grammar, which is found in the file `Doc/ls13_1.y`, for version 3.1, and `Doc/ls13_2.y`, for version 3.2.)

As a rough guide, one can have the following sections, in the following order.

- a context section, consisting of: the several assumes clauses, several includes clauses, and several shorthand declarations (enumeration, tuple, or union), all of which are optional,
- several introduces sections (typically zero or one), in which the signatures of operators are declared,
- several asserts sections (typically zero or one), in which various properties of operators are specified, and
- and an optional implies section, in which intended consequences of the preceding are specified. The parts of this section are intended to be redundant. LP can be used to debug a specification by checking that they really follow from the rest of the trait (see [Guttag-Horning93], Chapter 7).

A good general introduction is found in [Guttag-Horning93], Chapter 4. See Section 2.8 [What is the difference between assumes and includes?], page 17, for more on the purposes of the assumes and includes clauses.

2.8 What is the difference between assumes and includes?

One way to think of this is that includes requests textual inclusion of the named traits, but assumes only says that all traits that include the trait being specified must include some trait or traits that makes the assumptions true. Hence includes is like `#include` in C or C++, but assumes is quite different.

A common reason to use assumes instead of includes is when you want to specify a trait with a sort parameter, and when you also need some operators to be defined on your sort parameters. For example, Guttag and Horning’s handbook trait Sequence (see [Guttag-Horning93], page 174), starts as follows:

```larch
Sequence(E, C): trait
  assumes StrictPartialOrder(>, E)
```

The assumes clause is used to specify that the sort parameter E must have an operator > that makes it a strict partial order. One way to view such assumptions is that they specify what is needed about the theory of a sort parameter in order to sort check, and make sense of the operators that apply to the sort parameter [Goguen84] [Goguen86].

One can also use assumes to state assumptions about operators that are trait parameters. For example, Guttag and Horning’s handbook trait Sequence (see [Guttag-Horning93], page 175), starts as follows:

```larch
PriorityQueue (:E,E -> Bool, E, C): trait
  assumes TotalOrder(E for T)
```

The assumes clause is used to specify that the parameter :E,E -> Bool must make E into a total order.
Another reason to use an **assumes** clause instead of an **includes** clause is to state that a trait is not self-contained, but is intended to be included in some context in which some other trait has been included. For example, Guttag and Horning’s handbook trait IntegerPredicates (see [Guttag-Horning93], page 164), starts as follows.

```
IntegerPredicates (Int): trait
  assumes Integer
```

The **assumes** clause is used to specify that this trait has to be used in a trait that has included (the theory of) the trait Integer.

See [Guttag-Horning93], Section 4.6, for more details.

### 2.9 How and when should I use a partitioned by clause?

You should use a **partitioned by** clause to identify (i.e., make equivalent) some terms that would otherwise be distinct in an LSL trait’s theory. A classic example is in the specification of sets, which includes the following **partitioned by** clause (see [Guttag-Horning93], page 166).

```
C partitioned by \in
```

In this example, C is the sort for sets, and \in is the membership operator on sets. This means that two sets, s1 and s2, are distinct (i.e., not equal) if there is some element e such that member(e, s1) is not equal to member(e, s2). Another way to look at this is that, you can prove that two sets are equal by showing that they have the exactly the same members. (The sets insert(1, insert(2, {})) and insert(2, insert(1, {})) are thus equal.) This property is what distinguishes a set from a list, because it says that order does not matter in a set.

The reason there is a separate clause in LSL for this is because of the implicit quantifiers involved. The **partitioned by** clause in the example above is more powerful than the following (see [Wing95], Section 5.1).

```
\forall e: E, s1, s2: Set
  ((e \in s1) = (e \in s2)) => s1 = s2
```

The reason is that the above code would allow you to conclude that \{1,2\} = \{2,3\}, because both have 2 as a member. (That is, one can let e be 2, and then the left hand side of the implication holds.) Historically, LSL did not have a way to write quantifiers inside its formulas, and so the **partitioned by** clause was necessary. With LSL version 3.1, however, one could write the following equivalent to the above partitioned by clause for Set. (See Section 2.24 [How do I write logical quantifiers within an LSL term?], page 38, for the meaning of \A.)

```
\forall e: E, s1, s2: Set
  (\A e (e \in s1) = (e \in s2)) => s1 = s2
```

However, the above formula will be harder to work with in LP than the equivalent **partitioned by** clause. This is because LP has special provision for using **partitioned by** clauses. Another reason to prefer the **partitioned by** clause is that it conveys intent more clearly than a general logical formula like the above. Thus it there are still good reasons to use a **partitioned by** clause in LSL.

2.10 How and when should I use a generated by clause?

You should use a **generated by** clause to specify that there can be no abstract values beside the ones that you are describing for a given sort. This is quite common for specifications of the abstract values of data types, but will not usually be the case for the abstract values of sort parameters (such as `E` in `Queue[E]`). A **generated by** clause can also be thought of as excluding “junk” (i.e., nonstandard values) from a type [Goguen-Thatcher-Wagner78]. For example, in the trait `AnswerTrait` (see Section 2.5 [What is a sort?], page 16), nothing in the specification prevents there being other elements of the sort `Answer` besides `yes`, `no`, and `maybe`. That is, if `x` has sort `Answer`, then the formula

\[ x = yes \lor x = no \lor x = maybe \]

is not necessarily true; in other words, it’s not valid. However, in the following trait the above formula is valid, because of the **generated by** clause. (This is stated in the trait’s **implies** section.)

```plaintext
AnswerTrait2: trait
   includes AnswerTrait
   asserts
      Answer generated by yes, no, maybe
   equations
      yes \neq no;
      yes \neq maybe;
      no \neq maybe;
   implies
      \forall x: Answer
      x = yes \lor x = no \lor x = maybe;
```

Yet another reason to use a **generated by** clause is to assert a rule of data type induction. This view is particularly helpful in LP, because it allows one to divide up a proof into cases, and to use induction. For example, when proving some property `P(x)` holds for all `x` of sort `Answer`, the **generated by** clause of the trait `AnswerTrait2` allows one to proceed by cases; that is, it suffices to prove `P(yes)`, `P(no)`, and `P(maybe)`.

A more typical induction example can be had by considering Guttag and Horning’s handbook trait `Set` (see [Guttag-Horning93], page 167). In this trait, `C` is the sort for sets. It has the following **generated by** clause (by virtue of including the trait `SetBasics`, see [Guttag-Horning93], page 166).

```plaintext
C generated by {}, insert
```

This clause says that the only abstract values of sort `C`, are those that can be formed from (finite) combinations of the operators `{}` and `insert`. For example, `insert(x, insert(y, {}))` is a set. This rules out, for example, infinite sets, and sets that are nonstandard. It does not say that each such term denotes a distinct set, and indeed some terms fall in the same equivalence class (see Section 2.9 [How and when should I use a partitioned by clause?], page 18). It also does not mean that other operators that form sets cannot be defined; for example, in the trait `Set` one can write `{x} \U \{y\}`. What it does say is that every other way to build a set, like this one, has to be equivalent to some use of `{}` and `insert`.

Let us consider a sample proof by structural induction in the trait `Set`. One of the implications in this trait is that
\[ \forall s : C \\
size(s) \geq 0 \]

To prove this, one must prove that it holds for all sets \( s \). Since, by the \texttt{generated by} clause, all sets can be formed from \( {} \) and \texttt{insert}, it suffices to prove two cases. For the base case, one must prove the following trivial result.

\[ size(\{\}) \geq 0 \]

For the inductive case, one gets to assume that

\[ size(y) \geq 0 \]

and then one must prove the following.

\[ \forall s : C, e : E \\
size(insert(e, y)) \geq 0 \]

This follows fairly easily; we leave the details to the reader.

See section 4.2 in Guttag and Horning’s book [Guttag-Horning93] for more details on the \texttt{generated by} clause.

One situation where one usually does not want to use a \texttt{generated by} clause is when specifying the theory of a sort parameter. For example, in the trait \texttt{Set} (see [Guttag-Horning93], page 167), there is no \texttt{generated by} clause describing the elements \( E \) of the set. The reason not to use a \texttt{generated by} clause for such sort parameters is to make the assumptions on such parameters as weak as possible. Thus, unless one has to do induction over the elements of a sort parameter, one would not need to know that they are generated.

2.11 When should I use an \texttt{implies} section?

You should usually use an \texttt{implies} section in a trait, even though it adds nothing to the theory of a trait, and is completely redundant. The reasons for doing so are as follows.

- You probably have something interesting to highlight about the theory of a trait, to bring to the attention of the reader. This includes important examples that help illustrate how to use the operators.
- You probably have found more than one way to specify something in your trait. Putting both of them in the \texttt{asserts} section runs the risk of making your theory inconsistent (if they are not really equivalent). Leaving one of them out loses information, and prevents possible debugging and consistency checking that could otherwise be done (for example, using LP; see [Guttag-Horning93], Chapter 7).

Redundancy is a good thing for debugging and consistency checking; after all, one needs two pieces of information to do any checking.

A motto that mathematicians live by is the following. (You do realize that writing an LSL trait is doing math, don’t you?)

\begin{center}
Few axioms, many theorems.
\end{center}

For LSL, this motto means that you should try to make your \texttt{asserts} section as small and as elegant as possible, and to push off any redundancy into the \texttt{implies} section.

An important piece of redundant information you can provide for a reader in the \texttt{implies} section is a \texttt{converts} clause. You should also consider using a \texttt{converts} clause for each
Writing a good implies section for a trait is something that takes some practice. It helps to get a feel for what can be checked with LP first (see [Guttag-Horning93], Chapter 7). It also helps to look at some good examples; for example, take a look at the traits in Guttag and Horning’s handbook (see [Guttag-Horning93], Appendix A); most of these have an implies section, and some are quite extensive.

One general hint about writing implies clauses comes from the fact there are an infinite number of theorems that follow from every LSL specification. (One infinite set of theorems concerns the built-in sort Bool.) Since there are an infinite number of theorems, you can’t possibly list all of them. Focus on the ones that you think are interesting, especially those that will help the reader understand what your specification is about, and those that will help you debug your specification.


2.12 How and when should I use a converts clause?

A converts clause goes in the implies section of a trait. It is a way to make redundant claims about the completeness of the specification of various operators in a trait.

You should consider stating a converts clause for each non-constant operator you introduce in a trait. If you list an operator in a converts clause, you would be saying that the operator is uniquely defined “relative to the other operators in a trait” (see [Guttag-Horning93], p. 142, [Leavens-Wing97], Section B.1). Of course, this would not be true for every operator you might define. This means that if you state that an operator is converted in the implies section of a trait, then you are claiming that there is no ambiguity in the definition of that operator, once the other operators of the trait are fixed.

As an example of a converts clause, consider the following trait.

```
ConvertsExample: trait
  includes Integer
  introduces
    unknown, hmm, known: Int -> Int
  asserts \forall i: Int
    hmm(i) == unknown(i);
    known(i) = i + 1;
  implies
    converts
      known: Int -> Int,
      hmm: Int -> Int
```

In this trait, the converts clause claims that both known and hmm are converted. It should be clear that known is converted, because it is uniquely-defined for all integers. What may be a bit puzzling is that hmm is also converted. The reason is that once the interpretation of the other operators in the trait are fixed, in particular once the interpretation of unknown is fixed, then hmm is uniquely-defined. It would not be correct to claim that both hmm and
unknown are converted; since they are defined in terms of each other, they would not be uniquely defined relative to the other operators of the trait.\footnote{Although one cannot list both unknown and hmm in the same converts clause, one could add another converts clause to the trait ConvertsTrait, stating that unknown is converted, relative to the other operators, which in that case include hmm.}


Often one wants to say that some operator is uniquely-defined, but not for some arguments. This can be done using an exempting clause, which is a subclause of a converts clause. See Section 2.13 [How and when should I use an exempting clause?], page 22, for more discussion.

2.13 How and when should I use an exempting clause?

An exempting clause is a subclause of a converts clause. (See Section 2.12 [How and when should I use a converts clause?], page 21.) You should use an exempting clause when you want to claim that some operator is uniquely-defined, “relative to the other operators in a trait” ([Guttag-Horning93], p. 142), except for some special cases.

A very simple example is the trait List (see [Guttag-Horning93], p. 173), where the converts includes the following.

```plaintext
clears head
exempting head(empty)
```

This just says that head is well-defined, except for taking the head of an empty list.

A slightly more complicated exempts clause is illustrated by the following classic example. Suppose one specifies a division operator on the natural numbers. In such a trait (see [Guttag-Horning93], p. 201), one could use the following trait outline.

```plaintext
Natural (N): trait
  introduces
  0: -> N
  div: N, N -> N
    % ...
  asserts
    % ...
  implies
    converts div: N, N -> N
      exempting \forall x: N
        div(x, 0)
```

The converts clause in this example claims that div is uniquely-defined, relative to the other operators in a trait, except that division by 0 is not defined. Without the exempting subclause, the converts clause would not be correct.

See the following subsections and page 44 of [Guttag-Horning93] for more information.

2.13.1 Does exempting some term make it undefined?

No, exempting a term does not make it undefined. An exempting clause merely takes away the claim that it is defined (see [Guttag-Horning93], page 44).
List (see [Guttag-Horning93], p. 173), writing the following does not make head(empty) undefined.

converts head
   exempting head(empty)

If anything, it is the lack of an axiom defining the value of head(empty) that makes it “undefined”. However, you would certainly want to use an exempting clause in situations where you have a term that is intended to be “undefined”.

The concept of “undefinedness” is a tricky one for LSL. In a technical sense, nothing is undefined in LSL, because each constant and variable symbol has a value, and each non-constant operator is a total function (see [Guttag-Horning93], p. 9, and [Leavens-Wing97], Appendix A). It would be more accurate to say that in the trait List, the term head(empty) is “underspecified”. This means it has a value, but the trait doesn’t specify what that value is. See Section 2.16 [Is there support for partial specifications in LSL?], page 25, for more discussion on this point.

2.13.2 How can I exempt only terms that satisfy some condition?

With LSL (through version 3.1) the exempting subclause of a converts clause is not based on semantics, instead it is based on syntax. You can only exempt terms having particular forms; these forms allow the arguments to an operator to be either ground terms (as in head(empty)), or universally quantified variables or both (as in div(x, 0)).

If you want to exempt all terms for which one argument is positive, or has a size larger than 3, then you are out of luck. All you can do is to state your conversion claim informally in a comment (as you can’t use a converts clause naming the operator in question). If you want to rewrite your trait, you could, for example, change the appropriate argument’s sort to be the natural numbers instead of the integers, or define a new sort for which your operator is completely-defined.

See Section B.2 of [Leavens-Wing97] for a proposed extension to LSL that would solve this problem.

2.14 What is the meaning of an LSL specification?

An LSL trait denotes a theory, which is a collection of true formulas (of sort Bool). This theory contains “the trait’s assertions, the conventional axioms of first-order logic, everything that follows from them, and nothing else” (see [Guttag-Horning93], p. 37). For a brief introduction to these ideas, see Chapter 2 of [Guttag-Horning93]; for general background on equational logic, see [Ehrig-Mahr85] or [Loeckx-Ehrich-Wolf96].

Another way to look at the meaning of an LSL trait is the set of models it denotes. An algebraic model of a trait has a set for each sort, and a total function for each operator. In addition, it must satisfy the axioms of first-order logic and the axioms of the trait. A model satisfies an equational axiom if it assigns the same value to both sides of the equation. (See [Ehrig-Mahr85], Chapter 1, or [Loeckx-Ehrich-Wolf96], Chapter 5, for more details.)

In contrast to some other specification languages, the semantics of LSL is loose (see [Guttag-Horning93], p. 37). That is, there may be several models of a specification that have observably distinct behavior. For example, consider the trait ChoiceSet in Guttag
and Horning’s handbook (see [Guttag-Horning93], p. 176). There are many models of this trait with different behavior; for example, it might be that in one model \( \text{choose}(\{2\} \cup \{3\}) = 2 \), and in some other model \( \text{choose}(\{2\} \cup \{3\}) = 3 \). The specification won’t allow you to prove either of these things, of course, because whatever is provable from the specification is true in all models.

### 2.15 How does LSL handle undefined terms?

The following answer is adapted from a posting to ‘comp.specification.larch’ by Horning (July 19, 1995) in response to a question by Leavens (July 18, 1995).

The trouble starts with your question: there are no “undefined terms” in LSL. LSL is a language of total functions. There are no partial functions associated with operators, although there may be underspecified operators (i.e., operators bound to functions with different values in different models). The possibility of underspecification is intentional; it is important that models of traits do not have to be isomorphic.

Another way to say this is that in an algebraic model of a trait (see Section 2.14 [What is the meaning of an LSL specification?], page 23), each term denotes an equivalence class, which can be thought of as its definition in that algebra. In other algebras, the same term may be in a nonisomorphic equivalence class, if the term is underspecified.

To make this clearer, consider the following trait.

```
QTrait : trait
   includes Integer
   Q union of theBool: Bool, theInt: Int
   introduces
      isPositive: Q -> Bool
   asserts
      \forall q: Q
         isPositive(q) == q.theInt > 0 \&\& (tag(q) = theInt);
   implies
      \forall q: Q, b: Bool
         isPositive(q) == (tag(q) = theInt) \&\& q.theInt > 0;
         isPositive(q) == if (tag(q) = theInt)
            then q.theInt > 0 else false;
         isPositive(theBool(b)) == false;
   converts isPositive: Q -> Bool
```

In this trait, the question is whether the assertion that defines \text{isPositive} is written correctly. In particular, is \text{isPositive(theBool(true))} “undefined” or an error? This is clarified in the \text{implies} section of the trait, which claims that the order of conjuncts in the definition of \text{isPositive} does not matter, that the definition using \text{if} is equivalent, that the meaning of \text{isPositive(theBool(true))} is false, and that \text{isPositive} is converted (see Section 2.12 [How and when should I use a converts clause?], page 21).

The trait QTrait says that \text{q.theInt} is always of sort \text{Int}. However, it places no constraint on \text{what} integer it is, unless \text{tag(q) = theInt}. So in the definition and the first implication, both operands of \&\& are always defined (i.e., always mean either \text{true} or \text{false}, since \text{Bool} is generated by \text{true} and \text{false}), even though we may not always know which. So the definition is fine, and the implications are all provable.
The fact that you don’t have to use if, and that all the classical laws of logic (like conjunction being symmetric) apply is a big advantage of the total function interpretation of LSL. See Section 2.16.1 [Can I specify a partial function in LSL?], page 25 for more discussion.

2.16 Is there support for partial specifications in LSL?

There are two ways one might understand this question. These are dealt with below.

2.16.1 Can I specify a partial function in LSL?

Technically, no; all functions specified in LSL are total (see Section 2.14 [What is the meaning of an LSL specification?], page 23). Thus every operator specified in LSL takes on some value for every combination of arguments. What you can do is to underspecify such an operator, by not specifying what its value is on all arguments. For example, the operator head in the handbook trait List (see [Guttag-Horning93], p. 173) is underspecified in this sense, because no value is specified for head(empty). It is a synonym to say that head is weakly-specified.

One reason LSL takes this approach to dealing with partial functions is that it makes the logic used in LSL classical. One doesn’t have to worry about non-classical logics, and “bottom” elements of types infecting a term [Gries-Schneider95]. Another reason is that when LSL is used with a BISL, the precondition of a procedure specification can protect the operators used in the postcondition from any undefined arguments [Leavens-Wing97].

(If you want to develop a theory of partial functions, however, you can do that in LSL. For example, you can specify a type with a constant bottom, that, for you, represents “undefined”. You will just have to manipulate your partial functions with LSL operators that are total functions.)

2.16.2 Do I have to specify everything completely in LSL?

No, you don’t have to specify everything completely in LSL. It’s a good idea, in fact, to only specify the aspects that are important for what you are doing. For example, if you want to reason about graph data structures, it’s best to not (at first, anyway) try to specify the theory of graphs in great detail. Put in what you want, and go with that. (This won’t hurt anything, because whatever you need to prove you’ll be forced to add eventually.)

Anything you can specify in LSL, you can do incompletely. (There are no “theory police” that will check your traits for completeness.) To incompletely specify a sort, for example, nodes in a graph, just use it’s name in operations and don’t say anything more about it. If you need to have some operators that manipulate nodes, then write their signatures into the introduces section, but don’t write any axioms for them. If you need to know some properties of these operators, you can write enough axioms in the asserts section to prove them (perhaps making some of the properties theorems in the implies section).

The time to worry about the completeness of your theory is when you are having trouble proving some of the things you want to prove about it, or if you are preparing it for reuse by others. (On the other hand, a less complete theory is often more general, so when trying to generalize a trait for others to use, sometimes it helps to relax some of the properties you have been working with.)
2.17 Can I define operators using recursion?

Yes, you can define operators recursively. A good, and safe, way to do this is to use structural induction (e.g., on the generators of a sort). For example, look at how \texttt{count} is defined in the handbook trait \texttt{BagBasics} (see [Guttag-Horning93], p. 168). In this trait, the sort \texttt{C} of bags is generated by the operators \texttt{[]} and \texttt{insert}. There is one axiom that specifies \texttt{count} when its second argument is the empty bag, and one that specifies \texttt{count} when its argument is constructed using \texttt{insert}. There are many other examples in the Guttag and Horning’s handbook (see [Guttag-Horning93], Appendix A), for example the many specifications of operator \texttt{size} (Section A.5), the specification of the operators \texttt{head} and \texttt{tail} in the trait \texttt{Queue} (p. 171), the specification of \texttt{flatten} in the trait \texttt{ListStructureOps} (p. 183), and the specification of operator + in the trait \texttt{Positive} (p. 202). See Section 2.19 [Can you give me some tips for specifying things with LSL?], page 31, for more details on structural induction.

This kind of recursive specification is so common that if you are familiar with functional programming (especially in a language like Haskell or ML), then it’s quite helpful to think that way when writing LSL traits.

There is one thing to be careful of, however. Unless you use structural induction, you might specify an inconsistent trait. The following example (from David Schmidt, personal communication), illustrates the problem. This trait includes the \texttt{Integer} trait of Guttag and Horning’s handbook (see [Guttag-Horning93], p. 163).

\begin{verbatim}
BadRecTrait: trait
  includes Integer
  introduces
  f: Int -> Int
  asserts
  \forall i: Int
  f(i) == f(i) + 1;
\end{verbatim}

It’s easy to prove that this trait is inconsistent; subtract \( f(i) \) from both sides of the defining equation for \( f \), and you obtain \( 0 = 1 \). As a heuristic for avoiding this kind of mistake, avoid recursive specifications that don’t “make progress” when regarded as programs.

It’s also possible to fall into a pit when recursively defining a partial function. The following is a silly example, but illustrates the problem.

\begin{verbatim}
BadZeroTrait: trait
  includes Integer, MinMax(Int)
  introduces
  zero: Int -> Int
  asserts
  \forall k: Int
  zero(k) == if k = 0 then 0 else min(k, zero(k-1));
\end{verbatim}

This is silly, the idea being to define a constant function that returns zero for any non-negative integer. Still, it looks harmless. However, what does the specification say about \texttt{zero(-1)}. It’s easy to see that it’s less than \(-1\), and less than \(-2\), and indeed less than any integer. But the \texttt{Integer} trait in Guttag and Horning’s handbook (see [Guttag-Horning93], p. 163) does not allow there to be such an integer; so this trait is inconsistent. To solve this problem, you could change the axiom defining \texttt{zero} to indicate the intended domain, as follows.
∀ k: Int  
(k >= 0) => zero(k) = (if k = 0 then 0 else min(k, zero(k-1)));

This revised axiom would allow zero(-1) to have any value. Another cure for this kind of example would be to specify the domain of zero as the natural numbers. (Of course, for this particular example, a nonrecursive definition is also preferable.)

See Section 2.18 [What pitfalls are there for LSL specifiers?], page 27, for another thing to watch for when making a recursive definition.

2.18 What pitfalls are there for LSL specifiers?

According to Guaspari (posting to the larch-interest mailing list, on March 8, 1995), “the commonest ‘purely mathematical’ mistakes” in trait definitions occur when one uses structural induction “over constructors that don’t freely generate a sort”. To understand this, consider his example, which includes the handbook trait Set (see [Guttag-Horning93], page 167).

GuasparisExample: trait
includes Integer, Set(Int, Set[Int])

introduces
select: Set[Int] -> Int

asserts
∀ s: Set[Int], e: Int
select(insert(e, s)) == e;

implies
equations
insert(1, {2}) == insert(2, {1});
select(insert(1, {2})) == select(insert(2, {1}));
1 == 2; % ouch!

In the above example, the problem with the definition of select is that it’s not well-defined with respect to the equivalence classes of the sort Set[Int]. The problem is highlighted in the implies section of the trait. The first implication is a reminder that the order of insertion in a set doesn’t matter; that is, insert(1, {2}) and insert(2, {1}) are equivalent sets because they have the same members. In the jargon, insert(1, {2}) and insert(2, {1}) are members of the same equivalence class. The second implication is a reminder that an operator, such as select must give the same result on equal arguments; in the jargon, a function must “respect” the equivalence classes. But the definition of select does not respect the equivalence classes, because if one applies the definition to both sides of the second implication, one concludes that 1 = 2.

One way to fix the above definition is shown by the trait ChoiceSet in Guttag and Horning’s handbook (see [Guttag-Horning93], page 176). Another way that would work in this example (because integers are ordered) would be to define an operator minimum on the sort Set[Int], and to use that to define select as follows.

GuasparisExampleCorrected: trait
includes Integer, Set(Int, Set[Int]), MinMax(Int)
Chapter 2: The Larch Shared Language (LSL)

introduces
  select: Set[Int] -> Int
  minimum: Set[Int] -> Int

asserts
  \forall s: Set[Int], e, e1, e2: Int
  minimum(insert(e, {})) == e;
  minimum(insert(e1, insert(e2, s)))
  == min(e1, minimum(insert(e2, s)));
  select(insert(e, s)) == minimum(insert(e, s));

implies
  \forall s: Set[Int], e1, e2: Int
  minimum(insert(e1, insert(e2, s)))
  == minimum(insert(e2, insert(e1, s)));
  select(insert(1, {2})) == 1;
  select(insert(2, {1})) == 1;

To show that minimum is well-defined, one has to prove that it gives the same answer for all terms in the same equivalence class. The first implication is this trait does not prove this in general, but does help in debugging the trait.

To explain the difference that having a “freely-generated” sort makes, compare the trait GuasparisExample above to the handbook trait Stack (see [Guttag-Horning93], page 170). In that trait, top is defined using the following equation.

  \forall e: E, stk: C
  top(push(e, stk)) == e;

Why doesn’t this definition have the same problem as the definition of select? Because it respects the equivalence classes of the sort C (stacks). Why? Because there is no partitioned by clause for Stack, all equivalence classes of stacks have a canonical representative. The canonical representative is a term built using just the operators noted in the generated by clause for Stack: empty and push; it is canonical because there is only one way to use just these generators to produce a value in each equivalence class. The sort of stacks is called freely generated because of this property. (The sort for sets is not freely generated, because there is one term built using the generators that represents the value \{1,2\}.) Since a term of the form push(e, stk) is a canonical representative of its equivalence class, the specification of top automatically respects the equivalence classes.

A more involved example is the definition of the set membership operator (\in) for sets (see [Guttag-Horning93], p. 166 and p. 224). Since sets are not freely generated, it takes work to prove that the specification of \in is well-defined. You might want to convince yourself of that as an exercise.

In summary, to avoid this first pitfall, remember the following mantra when defining operators over a sort with a partitioned by clause or an interesting notion of equality.

\begin{center}
Is it well-defined?
\end{center}
As a help in avoiding this problem, you can use a generated freely by clause in the most recent versions of LSL. The use of a generated freely by clause makes it clear that the reader does not have to be concerned about ill-defined structural inductions.

Another pitfall in LSL is to assert that all values of some freely-generated type satisfy some nontrivial property. An example is the trait given below.

```
BadRational: trait
  includes Integer
  Rational tuple of num: Int, den: Int
  asserts
    \forall r: Rational
    r.den > 0;
  implies
    equations
    ([3, -4]).den > 0;
    (-4) > 0; % oops!
```

In this trait, the implications highlight the consequences of this specification pitfall: an inconsistent theory. The reason the above trait is inconsistent is that it says that all integers are positive. This is because for all integers i and j, one can form a tuple of sort rational by writing [i, j]. This kind of inconsistency arises whenever one has a freely generated sort, such as LSL tuples, and tries to add constraints as properties. There are several ways out of this particular pit. The most straightforward is to use a different sort for the denominators of rationals (e.g., the positive numbers, as in [Guttag-Horning93], page 207). Another way is to specify the sort Rational without using a tuple, but using your own constructor, such as mkRat; this allows the sort to be not freely generated. The only problem with that approach is to specify the sort Rational without using a tuple, but using your own constructor, such as mkRat; this allows the sort to be not freely generated. The only problem with that approach is that one either has to make arbitrary or difficult decisions (such as the value of mkRat(3, 0)), or one has to be satisfied with an underspecified constructor. Another way to avoid this is to use a parameterized trait; for example, we could have parameterized BadRational by the sort Int, and instead of including the trait Integer, used an assumes clause to specify the property desired of the denominator’s sort. In some cases, one can postpone asserting such properties until one reaches the interface specification; this is because in a BISL one can impose such properties on variables, not on entire sorts.

To avoid this pit, think of the following mantra.

```
Is this property defining a sort, or imposing some constraint on another sort?
```

(If the answer is that it’s imposing a constraint on another sort, watch out for the pit!)

A pitfall that some novices fall into is forgetting that LSL is a purely mathematical notation, that has no concept of state or side effects. For example, the following are surely incorrect signatures in LSL.

```
BadSig: trait
  includes Integer
  introduces
    currentTime: -> Seconds
    random: -> Int
```

If you think random is a good signature for a random number generator, you’ve fallen into the pit. The operator random is an (underspecified) constant; that is, random stands for
some particular integer, always the same one no matter how many times it is mentioned in an assertion. The operator `currentTime` illustrates how mathematical notation is “timeless”: it is also a constant.

How could you specify a notion like a clock in LSL? What you have to do is, instead of specifying the concept of “the current time”, specify the concept of a clock. For example consider the following trait.

```
ClockTrait: trait
    assumes TotalOrder(Seconds for T)
    introduces
        newClock: Seconds -> Clock
        tick: Clock -> Clock
        currentTime: Clock -> Seconds
    asserts
        Clock generated by newClock, tick
        \forall sec: Seconds, c: Clock
            currentTime(newClock(sec)) == sec;
            currentTime(tick(c)) > currentTime(c);
    implies
        \forall sec: Seconds, c: Clock
            currentTime(tick(tick(c))) > currentTime(c);
```

Note that a clock doesn’t change state in LSL, however, because each clock is also timeless, the `tick` operator produces a new `Clock` value. Tricks like this are fundamental to semantics (see, for example, [Schmidt86]). As an exercise, you might want to specify a random number generator along the same lines.

To avoid this pit remember the following mantra when you are writing the signatures of operators in LSL.

```
Does this operator have any hidden parameters?
```

There are no hidden parameters, or global variables, in LSL.

A logical pitfall that one can fall prey of in LSL is failing to understand that the quantifiers on an equation in LSL are outside the equation as a whole. Usually this problem can be detected when there is a free variable that appears only on the right-hand side of an equation. (This assumes that you use the typical convention of having the right-hand side of an equation be the “result” of the left-hand side.) For example, consider the following (from [Wing95], p. 13).

```
BadSubsetTrait: trait
    includes SetBasics(Set[E] for C)
    introduces
        __ \subset__ __: Set[E], Set[E] -> Bool
    asserts
        \forall e: E, s1, s2: Set[E]
        s1 \subset s2 == (e \in s1 => e \in s2);
    implies
        \forall e: E, s1, s2: Set[E]
        (e \in s1 /\ e \in s2) => s1 \subset s2;
```
The implication highlights the problem: if two sets have any single element in common, then they are considered subsets of each other by the above trait. This trait could be corrected in LSL by writing the quantifier inside the defining equation. (See Section 2.24 [How do I write logical quantifiers within an LSL term?], page 38, for the meaning of $\forall$.)

\[ \forall e: E, s1, s2: \text{Set}[E] \]
\[ s1 \subseteq s2 == \forall e (e \in s1 => e \in s2); \]

This is different, because the right-hand side must hold for all $e$, not just some arbitrary $e$.

To avoid this pit remember the following mantra whenever you use a variable on only one side of an equation.

\[
\text{Is it okay if this variable has a single, arbitrary value?}
\]

See Wing’s paper “Hints to Specifiers” [Wing95] for more tips and general advice.

2.19 Can you give me some tips for specifying things with LSL?

Yes indeed, we can give you some tips.

The first tip helps you write down an algebraic specification of a sort that is intended to be used as an abstract data type (see [Guttag-Horning78], Section 3.4, and [Guttag-Horning93], Section 4.9). The idea is to divide the set of operators of your sort into constructors and observers. A constructor returns your sort, while an observer takes your sort in at least one argument and returns some more primitive sort (such as $\text{Bool}$). The tip (quoted from [Guttag-Horning93], p. 54) is to:

Write an equation defining the result of applying each observer to each generator.

For example, consider the following trait.

ResponseSigTrait: trait
\[ \text{includes ResponseSigTrait} \]
\[ \text{asserts} \]
\[ \text{equations} \]
\[ \text{optimistic(yes)} == \text{true}; \]
\[ \text{optimistic(no)} == \text{false}; \]
implies
\forall r: \text{Response}
    yes \neq no;
    yes \neq maybe;
    no \neq maybe;
    optimistic(r) == r = yes \lor r = maybe;
    pessimistic(r) == r = no \lor r = maybe;

(Defining the operators optimistic and pessimistic as in the last two equations in the implies section of the above trait is not wrong, but doesn’t follow the tip. That kind of definition can be considered a shorthand notation for writing down the equations in the asserts section.)

As in the above example, you can specify that some set of generators is complete by listing them in a generated by clause. If you have other generators, what Guttag and Horning call extensions, then they can be defined in much the same way as the observers are defined. As an exercise, try defining the operator introduced in the following trait.

\text{ResponseTraitExercise}: \text{trait}
\quad \text{includes ResponseTrait}
\quad \text{introduces}
\quad \text{negate}: \text{Response} \rightarrow \text{Response}

Here are some other general tips for LSL.

- Try to find a trait that you can reuse before starting to write your own trait. Familiarity with Guttag and Horning’s handbook ([Guttag-Horning93], Appendix A) is very helpful. See Section 2.23 [Where can I find handbooks of LSL traits?], page 37 for other handbooks of traits.

- Break your trait into small pieces. While it’s not necessary to do so, your traits will be more reusable and more easily understood if they are small and focus on a tightly-related set of operators. One suggestion is to try to organize your trait as either an ADT (defining a sort such as sets, queues, integers, etc.), or as a mixin. A mixin trait defines a few operators, or a few properties on operators. Examples of mixin traits include all of those in Sections A.10 to A.14 of [Guttag-Horning93] (such as \text{Associative}, \text{Symmetric}, \text{TotalOrder}, and \text{MinMax}). A mixin trait typically does not have a generated by or partitioned by clause (although the latter sometimes are implied by such traits), whereas an ADT trait typically will have at least a generated by clause.

- When writing assertions, think about properties instead of algorithms. Since LSL specifications are not executed, you don’t have to worry about efficiency; instead worry about clarity and correctness. For example, when specifying a sort operator, the following is a perfectly good idea, even though the implicit algorithm (generate and test) is terribly inefficient.

\forall l1, l2: \text{List[OrderedItem]}
    \text{sort}(l1) = 12 == \text{permutation}(l1, 12) \lor \text{ordered}(12);
State implications in your traits, and use LP to debug your traits. Even if you don’t have time to do a lot of this, the mere act of thinking about whether what you have written really does imply what you think it should will have a beneficial effect on your traits.

When you use a negative number in a trait, always put parentheses around it. For example, the term \(-1 < 0\) does not parse in LSL unless you put parentheses around the \(-1\).

See Wing’s paper “Hints to Specifiers” [Wing95] for several more tips and general advice. You can also find other tips by looking for the “mottos” and “mantras” in this chapter. See Section 2.18 [What pitfalls are there for LSL specifiers?], page 27, for tips on what to avoid.

### 2.20 How do I know when my trait is abstract enough?

This is a good question, although there’s no hard-and-fast answer to it. To help you in deciding whether a specification is sufficiently abstract, we’ll give you a checklist, and an example of what not to do.

- Are you using an overly-specific representation?
- Are you specifying an algorithm for an operator instead of its properties?
- Are you working with an overly-specific sort of data instead of a sort parameter and some assumed properties?
- Are you specifying any unnecessary sizes or other constants?

To illustrate this checklist, we’ll give a really awful example of a specification that isn’t abstract enough. It illustrates every fault in the above list. (Of course, you’d never write anything this bad.)

```larch
definition NonAbstractSet: trait
   includes Integer
   SmallSet tuple of firstDef, secondDef, thirdDef: Bool,
   first, second, third: Int
   introduces
   {}: -> SmallSet
   insert: Int, SmallSet -> SmallSet
   __ \in __: Int, SmallSet -> Bool
   size: SmallSet -> Int
   asserts
   \forall i,i1,i2,i3: Int, s: SmallSet, b1,b2,b3: Bool
   {} == [false, false, false, 0, 0, 0];

   insert(i, s)
   == if ~s.firstDef
       then set_first(set_firstDef(s, true), i)
       else if ~s.secondDef
```
then set_second(set_secondDef(s, true), i)
else if ~s.thirdDef
    then set_third(set_thirdDef(s, true), i)
else s;

i \in s == if s.firstDef ∧ i = s.first
    then true
    else if s.secondDef ∧ i = s.second
        then true
        else if s.thirdDef ∧ i = s.third
            then true
            else false;

size(s) == if ~s.firstDef then 0
else if ~s.secondDef then 1
else if ~s.thirdDef then 2
else 3;

Comparing this specification with the one in Guttag and Horning’s handbook ([Guttag-Horning93], pp. 166-167) reveals a lot about how to write more abstract specifications. Notice that the trait NonAbstractSet represents the sort SmallSet as a tuple. This is akin to a “design decision” and makes the specification less abstract. Another problem is the specification of algorithms for the observers insert, \in, and size. These could be made more abstract, even without changing the definition of the sort SmallSet. For example, using the idea of generators and observers (see Section 2.19 [Can you give me some tips for specifying things with LSL?], page 31), one could specify \in as follows.

\begin{verbatim}
i \in {} == false;
size(s) < 3 => (i \in insert(i, s) = true);
\end{verbatim}

The above trait also defines sets of integers, even though it doesn’t particularly depend on the properties of integers. It would be more abstract to specify sets of elements. Finally the trait only works for sets of three elements. Unless there is some good reason for this (for example, modeling some particular implementation), you should avoid such arbitrary limits in LSL. They can easily be imposed at the interface level.

Look at traits in the LSL handbook for examples of how to do things right. You may find that a trait can be too abstract to make it easily understood; that’s fine, you don’t have to try to define the ultimate theory of whatever you’re working with.

See [Wing95], section 4.1 for a more extended discussion on the above ideas. You might want to look at [Liskov-Guttag86] for more background on the general idea of abstraction in specification. See also [Jones90] for the related idea of “implementation bias”.

2.21 Is there a way to write a trait that will guarantee consistency?

The question of how to write a trait that will guarantee consistency is important for two reasons.

- If you use all of LSL, which includes first-order logic, consistency is undecidable.
- If your trait is inconsistent, then for most purposes the theory you are specifying is worthless, because everything will be provable (even equations such as true == false).
You can trade off ease of expression vs. ease of proving consistency as follows (quoted from Garland’s posting to ‘comp.specification.larch’, June 26, 1996).

If you are willing to sacrifice ease of proving consistency, you can axiomatize your theories in first-order logic, where consistency is undecidable.

If you are willing to sacrifice ease of expression, you can achieve consistency as do Boyer and Moore [Boyer-Moore79][Boyer-Moore88] with their “principle of definition”; that is, you can axiomatize your theories in primitive recursive arithmetic rather than in first-order logic.

If you are willing to settle for some sort of middle ground, and you are willing to wait, a future release of the LSL Checker may provide some help. I’ve been thinking about how to enhance the checker to provide warnings about potentially inconsistent constructions, basically by using a less restrictive version of Boyer-Moore principle of definition to classify constructs as safe or unsafe.

Part of my envisioned project for having LSL issue warnings about potential inconsistencies involves generating proof obligations for LP that can be used to establish consistency when it is in doubt.

So you can understand this tradeoff better, and so you can be aware of when to worry about consistency, we will explain Boyer and Moore’s shell principle and definition principle and how they apply to LSL.

Boyer and Moore’s shell principle (see [Boyer-Moore88] Section 2.3 and [Boyer-Moore79] Section III.E) is a way of introducing new tuple-like sorts; one would follow it literally in LSL by only using tuple of to construct sorts. This prevents inconsistency, because it prevents the operators that construct the new sorts from changing the theory of existing sorts. (That is, the new theory is a conservative extension of the existing theory; see, for example, Section 6.12 of [Ehrig-Mahr85].)

Although following the shell principle literally seems far too restrictive in practice for LSL, one can try to use its ideas to define new sorts in a disciplined way to help prevent inconsistency. This is done by divide the set of operators of your sort into generators and observers and specifying the result of applying each observer to each generator, as described above (see Section 2.19 [Can you give me some tips for specifying things with LSL?], page 31). For an example, see the handbook trait SetBasics (in [Guttag-Horning93], page 166). However, it’s not clear whether following this discipline guarantees that your theory will be consistent.

Boyer and Moore’s definition principle (see [Boyer-Moore88] Section 2.6) is a way of defining auxiliary functions (operators in a trait that are not generators and that are not the primitive observers used in defining a sort). The definition principle does not allow one to underspecify a function; its value must be defined in all of its declared domain. Furthermore, recursion is only allowed if one can prove that the recursion is terminating. For LSL this means that a definition of operator f satisfies the definition principle if it satisfies the following conditions.

• There is just one equation that defines the operator f, and it has the following form, where x1, ..., xn are distinct logical variables, called the “formals”, and body is an LSL term, called the “body”.

\[
\text{f}(x_1, \ldots, x_n) \equiv \text{body}
\]

• The only free variables in body are the formals, x1, ..., xn.
• If the definition is recursive, then the recursion must be proved to terminate.

See Section 2.17 [Can I define operators using recursion?], page 26, for how a nonterminating recursive definition can cause inconsistency.

For examples of what may happen when some of the other conditions of the definition principle are violated, consider the following trait.

BadDefinitionsTrait: trait
  includes Integer
  introduces
    moreThanOneAxiom: Int -> Bool
    freeVarsInBody: Int -> Int
  asserts
    \forall i, k: Int
      moreThanOneAxiom(i) == i >= 0;
      moreThanOneAxiom(i) == i <= 0;
      freeVarsInBody(i) == k + i;
  implies
    equations
      true == false;
      3 == 2;

The two axioms for the operator moreThanOneAxiom makes the theory inconsistent; to see this, consider the following calculation.

true
  = {by the trait Integer}
  4 >= 0
  = {by the first axiom}
  moreThanOneAxiom(4)
  = {by the second axiom}
  4 <= 0
  = {by the trait Integer}
  false

See Section 2.18 [What pitfalls are there for LSL specifiers?], page 27, for the trait BadRational that illustrates a more subtle way to fall into this kind of inconsistency.

In the trait BadDefinitionsTrait above, the axiom for freeVarsInBody also makes the theory inconsistent. This can be seen as follows.

3
  = {by the trait Integer}
  3 + 0
  = {by the axiom for freeVarsInBody, with k = 3}
  freeVarsInBody(0)
  = {by the axiom for freeVarsInBody, with k = 2}
  2 + 0
  = {by the trait Integer}
  2

See Section 2.18 [What pitfalls are there for LSL specifiers?], page 27, for the trait BadSig that illustrates another way to fall into this kind of inconsistency.

Because Boyer and Moore’s definition principle is quite restrictive, it should be emphasized that the definition principle does not have to be used in LSL, and is not even necessary
for making consistent definitions. For example, it seems perfectly reasonable to specify an
auxiliary operator by using several equations, one for each generator of an argument sort
(see Section 2.19 [Can you give me some tips for specifying things with LSL?], page 31),
instead of using just one equation and \textbf{if then else}. The definition principle requires that all
the variables in the defining equation be distinct, but this seems only to prevent underspec-
ification, not inconsistency. (See Section 2.16.2 [Do I have to specify everything completely
in LSL?], page 25, for a discussion of underspecification.) The definition principle also
prohibits mutual recursion.

Remember that it’s a tradeoff: if you use the definition principle, you are guaranteed
to make consistent auxiliary function definitions, but it may be harder to specify what you
want, and your specifications may be less clear than they would be otherwise.

See Section 3.17 [How do I prove that my theory is consistent with LP?], page 48, for
how to use LP to help prove that a trait is consistent. This would be useful if the trait
violates either Boyer and Moore’s shell principle or the definition principle.

2.22 Do I have to write all the traits I am going to use from
scratch?

No, you don’t have to write all the traits you are going to use from scratch. You
should certainly try to use Guttag and Horning’s LSL handbook traits ([Guttag-Horning93],
Appendix A), which are freely available. Others have also made other handbooks available.
See Section 2.23 [Where can I find handbooks of LSL traits?], page 37, for details on how
to get these handbooks.

The whole idea of LSL is to enable reuse of traits, so you should familiarize yourself with
the relevant handbooks before starting to write new traits.

2.23 Where can I find handbooks of LSL traits?

The most commonly-used handbook of LSL traits is Guttag and Horning’s handbook
([Guttag-Horning93], Appendix A). This can be obtained by anonymous ftp with the LSL
checker (see Section 2.3 [Where can I get a checker for LSL?], page 15). A hypertext version
is on-line in


A general resource for all known handbooks that are publically available is found on the
world-wide web, at the following URL.

http://www.cs.iastate.edu/~leavens/Handbooks.html

Besides a pointer to Guttag and Horning’s handbooks, other handbooks found there include
handbooks of: mathematical traits, calendar traits, Larch/C++ traits. There is also a
handbook of “SPECS traits”, which allows one to specify in a style similar to VDM-SL.
The LSL input for these handbooks can be obtained from the world-wide web page given
above, or by anonymous ftp from the following URL.

2.24 How do I write logical quantifiers within an LSL term?

In LSL 3.1, you can write a universal quantifier within an LSL term by using \( \forall \), and an existential quantifier using \( \exists \). As an example, consider the following trait (from Leavens's Math handbook).

\[
\text{DenseOrder}(T) : \text{trait} \\
\quad \text{includes PartialOrder} \\
\quad \text{asserts} \\
\quad \forall x,y,z : T \\
\quad \quad (x < y) \Rightarrow (\exists z (x < z \land z < y)); \\
\quad \text{implies} \\
\quad \forall x,y,z,t,u : T \\
\quad \quad (x < y) \Rightarrow (\exists z \exists t (x < z \land z < t \land t < y)); \\
\quad \quad (x < y) \Rightarrow (\exists z \exists t \exists u (x < z \land z < t \land t < u \land u < y));
\]

Note that the variables used in quantifiers are declared before the term. The variable \( z \) used in the asserts section is existentially quantified, and not universally quantified as might appear from its declaration.

2.25 Where can I find LaTeX or TeX macros for LSL?

You can get a LaTeX style file, ‘larch.sty’, and a macro file defining a bunch of mathematical symbols, ‘larchmath.tex’, by anonymous ftp from the following URL.

\texttt{ftp://ftp.cs.iastate.edu/pub/larch/tex}

These files were originally designed by Lamport, and revised by Horning.

The documentation for ‘larch.sty’ says that it is to be used with LaTeX 2.09. However, it can be used with LaTeX2e. To do so, put the following lines at the start of your LaTeX input.

\[
\documentclass{article} \\
\usepackage{larch}
\]

Many published Larch papers and documents have formatting produced by these TeX macros. However, it’s not clear that the use of special symbols is a good thing. Some of us prefer to let readers see the way the input is typed (e.g., \( \in \)), because we think that non-mathematicians find it easier to understand words instead of arbitrary symbols. There is some experimental evidence for this [Finney96], so you might want to think about who your audience is before you go to a great effort to print fancy symbols in your specifications.

See Section 2.26 [How do I write some of those funny symbols in the Handbook in my trait?], page 38.

2.26 How do I write some of those funny symbols in the Handbook in my trait?

The ISO Latin codes for all of the “funny symbols” used in Guttag and Horning’s handbook are found on page 224 of [Guttag-Horning93]. Type the characters on the right hand side when using LSL. If you are reading the handbook in print form, you might want to tape a copy of that page to your wall.
See Section 2.25 [Where can I find LaTeX or TeX macros for LSL?], page 38, for details on how to get your specification to print out like the handbook.

### 2.27 Is there a literate programming tool for use with LSL?

Yes, there actually is a version of “spiderweb” specialized for use with LSL. If you are really a fan of such fancy systems, you can find it by using the literate programming library’s URL.

http://www.desy.de/user/projects/LitProg.html

(If you don’t know what literate programming is, you can find out there as well.)

However, we have found that using the “noweb” system is much easier for most people, and nearly as good. You can get noweb from the literate programming library (see above), or directly from the following URL.

http://www.eecs.harvard.edu/~nr/noweb/intro.html

Even easier, it’s possible to use the trait modularization facilities of LSL (includes, and assumes) to use virtually the same style as a literate programming system would support.

### 2.28 Is there a tool for converting LSL to hypertext for the web?

Yes, Penix’s “lsl2html” tool converts an LSL input file to HTML, so it can be browsed over the net. It can be found at the following URL.

http://www.ece.uc.edu/~kbse/lsl2html/

Unfortunately, Penix’s tool has a few problems that have never been fixed. Instead, you might want to use Leavens’s tool “lcpp2html”, which is available from the following URL.

http://www.cs.iastate.edu/~leavens/lcpp2html.html

To use this like “lsl2html” invoke it as follows.

```bash
lcpp2html -P -I *.lsl
```
3 The Larch Prover (LP)

In this chapter, we discuss questions about the larch prover (LP).

3.1 What is the Larch Prover (LP)?

The Larch Prover (LP) [Garland-Guttag95] is a program that helps check and debug proofs. It is not geared toward proving conjectures automatically, but rather toward automating the tedious parts of proofs. It automates equational rewriting (proofs by normalization), but does not (by default) automatically try other proof techniques.

Although LP is a general proof assistant, its main uses in the context of Larch are to:

- aid the debugging of specifications (i.e., theories), by helping develop redundant conjectures and their proofs, and
- checking old proofs in the context of changed theories, to see if the proof is still valid.

The philosophy behind LP is “based on the assumption that initial attempts to state conjectures correctly, and then to prove them, usually fail” ([Garland-Guttag91], page 1). Because of this, LP does not (by default) automatically attempt steps in a proof that are difficult (i.e., speculative, or mathematically interesting). Thus LP does not appear to be a “smart” assistant in finding proofs; it will not suddenly try a complex induction by itself. This prevents LP from spending lots of computer time on conjectures that are not true, and helps make LP predictable. LP can be thought of more as a “faithful” and “exacting” assistant. This is appropriate in the debugging stage of a proof, where the steps you should follow in interacting with LP are as follows.

- LP applies equational rewriting to your conjecture, and either tells you that it’s true, or presents you with a subgoal that, if proved, will lead to the proof of your conjecture.
- If you have something left to prove, then ask yourself: “why do I think this is true?”
- If you don’t think the current subgoal is true, then you can change your conjecture (for example, by strengthening the hypothesis) or your theory, and start over.
- If you think the current subgoal is true, but think you’ve started the proof out the wrong way, then you can backtrack in the proof (by using the cancel command).
- If you think the current subgoal is true, then tell LP to carry out a step in the proof that corresponds to your reason. (For example, if you believe that the subgoal is true because it’s true in each of several cases, use the resume by cases command.)
- LP will then carry out the corresponding proof step for you, and this process starts over.

Think of using LP as a game whose prize is certainty about why your conjecture is true, or at least knowledge about what’s wrong with your theory or conjecture. The main tactical knowledge you need to play the game well is the correspondence between reasons and proof techniques (see Section 3.15 [Can you give me some tips on proving things with LP?], page 46). Also needed for good play is the ability to always question the consistency of your theory and the truth of your conjectures.

The interaction described above is fine for developing conjectures and their proofs, but LP also has features for replaying proofs, so that you don’t have to interact with it when
checking old proofs in slightly modified settings. See Section 3.20 [How can I develop a proof that I can replay later?], page 49

See [Guttag-Horning93], Chapter 7, and [Garland-Guttag95] for more information. See Section 3.2 [What kind of examples have already been treated by LP?], page 41, for more discussion about the uses of LP.

3.2 What kind of examples have already been treated by LP?

The original reason LP was built to help debug formal specifications (see [Garland-Guttag-Horning90] and Chapter 7 of [Guttag-Horning93]). It has, however, also been used for several other purposes, including the following (reported in messages to the larch-interest mailing list, February 7 and July 23, 1994).

- Formal verification of circuits and general reasoning about hardware designs (including proofs of invariants, properties, refinement, and equivalence between implementations and specifications, see, for example, [Saxe-etal92] [Staunstrup-Garland-Guttag89] [Staunstrup-Garland-Guttag92]),
- Reasoning about various mathematical theories, such as relational arithmetic and algebra [Martin-Lai92],
- Verifying sequential programs (such as compilers) or properties of such programs.
- Verification of concurrent programs (such as a distributed multiuser service) or properties of concurrent algorithms [Luchangco-etal94] [Soegaard-Anderson-etal93] and communication protocols [Chetali98].

Several other papers about uses of LP appear in [Martin-Wing93].

3.3 How does LP compare with other theorem provers?

The basic difference between LP and other theorem provers is that LP does not automatically attempt complex (i.e., interesting or speculative) proof steps. LP also has no way to write general proof strategies or tactics (as can be done in PVS, LCF, and other theorem provers). An additional convenience is that there are tools for supporting the translation of LSL specifications into LP’s input format.

Compared with PVS [Owre-etal95], LP has a less sophisticated user interface, and less automation. For example, PVS allows one to reorder what lemmas should be proved more easily. PVS also has decision procedures for ground linear arithmetic, and possibly better integration of arithmetic with rewriting. The language used by PVS also has a more sophisticated type system than does that of LSL (or LP), as it uses predicate subtyping and dependent types. PVS also provides a strategy language for expressing proof strategies, although it seems that users typically do not write their own strategies. PVS also has various syntactic and semantic features, such as support for tables and various kinds of model checking that are not available in LP. On the other hand, LP is more portable (runs on more kinds of computers), and because its interface does not rely on emacs, it requires less set-up. Furthermore, as a teaching aid, a less automatic user interface and a simpler type system have benefits in making students think more.
Chapter 3: The Larch Prover (LP)

The Boyer-Moore theorem prover, NQTHM [Boyer-Moore79] [Boyer-Moore88] is also more aggressive than LP in trying to actively search for proofs. Its successor, ACL2 is described as “industrial strength” [Kaufmann-Moore97].

A major difference between LP and HOL [Gordon-Melham93], Isabelle [Paulson94], and PVS is that LP does not use a higher-order logic, which the others do. Isabelle also has extensive support for term rewriting.

3.4 Where can I get an implementation of LP?

Currently, LP only runs on Unix machines. You can get an implementation of LP using the world-wide web, starting at the following URL.


You can also get an implementation by anonymous ftp from the following URL.


See Section 2.3 [Where can I get a checker for LSL?], page 15, for information about getting a checker for LSL (that translates LSL traits into LP’s input format), and also for information on the Larch Development Environment (LDE), that supports LSL, LP, LCL, and Larch/C++.

3.5 Is there a command reference or list of LP commands?

Yes, there is a summary of LP’s commands in the section titled “Command Summary” of [Garland-Guttag95]. For ease of reference, you can find this at the following URL.

http://www.sds.lcs.mit.edu/spd/larch/LP/commands/commands.html

(You might want to print this out and paste it on your wall.)

Within an LP process, you can see this by issuing the command help commands. Use help display to see help on the command display. Use display ? to see the arguments you can pass to the command display.

3.6 Can I change the erase character in LP?

According to Garland (posting on the larch-interest mailing list, July 14, 1995), the erase (or rubout) character used by LP cannot be changed easily. The reason is that “LP is written in CLU, and LP’s line editing features are provided by the CLU runtime environment. This environment hard-wires the delete key [DEL] as the rubout character.”

You can erase the current line in LP by typing a C-u (control-U). You can redisplay the current line by typing C-r. See [Garland-Guttag91], page 7 for more details.

See Section 3.20 [How can I develop a proof that I can replay later?], page 49, for a way to avoid tedium and much direct interaction with LP.

If you use emacs, there is a way to get completely around this problem. Simply start emacs, and type M-x shell, and then run LP from within the Unix shell that appears. Besides allowing arbitrary editing, this trick allows you to scroll backwards to view output.
3.7 How do I interrupt LP?

If you are running LP under Unix, the Unix “quit” character (usually C-g, that’s control-g) will interrupt LP’s execution.

See Section 3.6 [Can I change the erase character in LP?], page 42, for other characters that aid in editing interactive input to LP. See Section 3.20 [How can I develop a proof that I can replay later?], page 49, for a way to avoid tedium and much direct interaction with LP.

3.8 Do I need to use LSL if I use LP?

No, you do not need to use LSL if you use LP. LP has its own input format (although it is very similar to LSL’s input format). So, many users of LP simply bypass LSL, and use LP exclusively. On the other hand, using LSL as an input format to LP has the following advantages.

- The LSL checker automatically generates proof management commands (scripting and logging) for LP, and helps organize theories and conjectures into files. See Section 3.10 [How do I use LP to check my LSL traits?], page 43.
- Your theories and conjectures are accessible to a wider audience if they are stated in LSL format rather than in LP input format.

3.9 Do I need to use LP if I use LSL?

No, you don’t have to use LP if you use LSL. However, it’s very helpful to use LP with LSL, because it aids in debugging LSL traits. The idea is that you state redundant properties of the theory you are specifying in the \texttt{implies} section of your LSL trait (see Section 2.11 [When should I use an implies section?], page 20), and then use LP to check that those really do follow from the trait. See Section 3.10 [How do I use LP to check my LSL traits?], page 43, and Chapter 7 of [Guttag-Horning93], for how to do this.

3.10 How do I use LP to check my LSL traits?

There are three main things to check about an LSL trait (see [Guttag-Horning93], Chapter 7).

- Is the trait consistent?
- Do the trait’s assumptions (if any) hold?
- Do the traits implications hold?

The first kind of check is vital, because you can prove anything in an inconsistent theory. See Section 3.17 [How do I prove that my theory is consistent with LP?], page 48, and Section 7.6 of [Guttag-Horning93], for more information on how to do this.

The use of an \texttt{assumes} clause in a trait specifies assumptions about the context in which that trait is used (see Section 2.8 [What is the difference between assumes and includes?], page 17); if your trait has included other traits with \texttt{assumes} clauses, then these assumptions must be checked (see page 124-125 of [Guttag-Horning93]). However,
checking such assumptions is similar to the checking that happens when checking a trait’s implications, which is the focus of this section.

Here are the top-level steps in using LP to check the implications written in the `implies` section of your trait. Let us suppose your trait is called `Foo`, and is in a file named ‘`Foo.lsl`’.

- To produce LP input from your trait, you would use the ‘`-lp`’ switch with the LSL checker as follows.
  
  ```
  lsl -lp Foo.lsl
  ```

  This produces the following files: ‘`Foo_Axioms.lp`’, ‘`Foo_Checks.lp`’, and ‘`Foo_Theorems.lp`’.

- Start LP. The suggested name for this command is `lp`, but on many systems it is called `LP`.
  
  ```
  lp
  ```

- LP will prompt you for input with a prompt that looks like ‘`LP1: `’. When you see this prompt, type the following.
  
  ```
  execute Foo_Checks
  ```

- LP will read in your input and attempt the first (rewriting) steps of the proofs of your conjectures. Often it will suspend each proof waiting for your input. See Section 3.1 [What is the Larch Prover (LP)?], page 40, for the steps to follow in debugging your trait by trying to prove the conjectures in its `implies` section. See Section 3.15 [Can you give me some tips on proving things with LP?], page 46 for tips on how to go about a proof.

  See Section 3.20 [How can I develop a proof that I can replay later?], page 49, for an alternative to using LP interactively.

  See Section 3.15 [Can you give me some tips on proving things with LP?], page 46 and Sections 7.4-7.5 of [Guttag-Horning93], for more information on checking implications with LP.

### 3.11 What is the use of each kind of file generated by the LSL checker?

The LSL checker, when used with the ‘`-lp`’ switch on a file named ‘`Foo.lsl`’, in general produces three files: ‘`Foo_Axioms.lp`’, ‘`Foo_Checks.lp`’, and ‘`Foo_Theorems.lp`’. Each of these files is in LP input format, hence the suffix ‘`.lp`’.

The file ‘`Foo_Axioms.lp`’ contains a translation of the trait in ‘`Foo.lsl`’, minus the `implies` section. It is “executed” by LP when LP is executing the translation of some other LSL trait that includes `Foo`.

The file ‘`Foo_Checks.lp`’ contains a translation of the implications in ‘`Foo.lsl`’, along with checks of assumptions. As an LP command file, it starts logging and scripting to the appropriate files (‘`Foo.lplog`’ and ‘`Foo.lpscr`’), executes the ‘`Foo_Axioms.lp`’ file, and then attempts proofs for each of the checks. This is the main file you execute when using LP.

The file ‘`Foo_Theorems.lp`’ contains a translation of the implications in ‘`Foo.lsl`’, stated as axioms. It is used automatically when it is safe to do so in the LP input generated by the LSL checker.
All of these files are text files, so viewing them with an editor, or printing them out, is a good way to solidify your understanding. (See also page 128 of [Guttag-Horning93].)

3.12 If LP stops working on my input is it all correct?

No, if LP stops working, it may just mean that it wants more guidance from you. The way to tell if all outstanding conjectures have been proved is to use the `qed` command [Garland-Guttag95]. If you see the following, then you are done.

All conjectures have been proved.

However, if you see something like the following, then you are not done.

Still attempting to prove level 2 lemma FooTheorem.2

See Section 3.13 [How do I find out where I am in my proof?], page 45, for a way to find out what has to be done to finish your proof.

3.13 How do I find out where I am in my proof?

When LP stops reading input from your file (for example, the ‘_Checks.lp’ file for an LSL trait), the first thing you should do is to get your bearings. The commands described here also are helpful if you are using LP interactively, because it’s easy to forget what you are trying to do.

To find your bearings, the following commands are helpful (see the section titled “Sample Proofs: how to guide a proof” in [Garland-Guttag95]).

<table>
<thead>
<tr>
<th>To find out ...</th>
<th>Use the command</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is left to be proved?</td>
<td>display conjectures</td>
</tr>
<tr>
<td>What is the status of my proof?</td>
<td>display proof-status</td>
</tr>
<tr>
<td>How did I get here?</td>
<td>history</td>
</tr>
<tr>
<td>What axioms can I use?</td>
<td>display</td>
</tr>
<tr>
<td>What are the current hypotheses?</td>
<td>display *Hyp</td>
</tr>
<tr>
<td>What are the induction rules?</td>
<td>display induction-rules</td>
</tr>
<tr>
<td>What are the deduction rules?</td>
<td>display deduction-rules</td>
</tr>
</tbody>
</table>

The commands `display conjectures` and `display` seem to be the most useful. When you use `display conjectures`, LP shows where it stopped in your proofs. There may be several lemmas listed below the first “conjecture”. The last of these (the one with the greatest “level” number) is the one you can work on first.

See Section 3.15 [Can you give me some tips on proving things with LP?], page 46, for tips on how to proceed with your proof, once you’ve found your bearings.

3.14 What proof techniques does LP attempt automatically?

By default, LP automatically uses only the following forward inference techniques.

- normalization, and
- application of deduction rules (that are not inactive).

See Section 3.15 [Can you give me some tips on proving things with LP?], page 46, and Sections 7.4-7.5 of [Guttag-Horning93], for other proof techniques.
3.15 Can you give me some tips on proving things with LP?

The most important tip for using LP is to think about your conjectures, and to be very skeptical of their truth.

When you first start using LP, you will be tempted to have it do the thinking for you. You may find yourself trying random proof techniques to prove the current subgoal. Resist that temptation! Curb your desire for automatic proof! Instead, use one of the following two ideas (see the section titled “Sample Proofs: how to guide a proof” in [Garland-Guttag95]).

Think about whether the conjecture is true, and if so, why you believe it. (Imagine trying to convince a very skeptical, but logical, person.) If you believe your conjecture, then your reason may suggest a proof technique. Here is a small table relating reasons and some of LP’s proof techniques.

<table>
<thead>
<tr>
<th>If your argument has the shape ...</th>
<th>Then try the following ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>For a simple case ..., then assuming ...</td>
<td>resume by induction on for bigger cases...</td>
</tr>
<tr>
<td>On the one hand, on the other hand...</td>
<td>resume by cases ... or sometimes resume by induction on ...</td>
</tr>
<tr>
<td>If that weren’t true, then ...,</td>
<td>resume by contradiction sometimes followed by critical-pairs #Hyp with *</td>
</tr>
<tr>
<td>By axiom R, ...</td>
<td>apply R to conjecture or if R contains a free or universally quantified variable, x instantiate x by t in R or if R has an existentially quantified variable, x fix x as t in ...</td>
</tr>
<tr>
<td>This should simplify by ...</td>
<td>normalize conjecture with ... or if the ... is a formula rewrite conjecture with ...</td>
</tr>
</tbody>
</table>

Proofs by induction are very common and more useful than you might think. For example, induction is used to prove the last two implications in ResponseTrait (see Section 2.19 [Can you give me some tips for specifying things with LSL?], page 31); in that proof, there are three basis cases, and no inductive case. This shows that any time you have a generated sort, a proof by induction can be used, even if there are only a finite number of values of that sort. Furthermore, you should consider a proof by induction first, as some other proof techniques (by cases, =>, if-method, contradiction) make proof by induction more difficult or impossible, because they replace free variables by constants (see [Garland-Guttag91], page 65).
Another idea is to see if the form of your conjecture suggests a proof technique. Here is a small table of conjecture forms and proof techniques. (A name of the form fresh1 is to be replaced by a fresh variable name, that is, by a name that is not used elsewhere.)

<table>
<thead>
<tr>
<th>If your conjecture looks like ...</th>
<th>Then try the following ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>~(E1 = E2)</td>
<td>resume by contradiction</td>
</tr>
<tr>
<td></td>
<td>% sometimes followed by</td>
</tr>
<tr>
<td></td>
<td>critical-pairs *Hyp with *</td>
</tr>
<tr>
<td>(P1 \land P2 \land \ldots)</td>
<td>resume by (\land)</td>
</tr>
<tr>
<td>(P \Rightarrow Q)</td>
<td>resume by (\Rightarrow)</td>
</tr>
<tr>
<td>(P \Leftarrow Q)</td>
<td>resume by (\Leftarrow)</td>
</tr>
<tr>
<td>if (H) then (P) else (Q)</td>
<td>resume by if-method</td>
</tr>
<tr>
<td>(\ldots(\text{if }P\text{ then }\ldots)\ldots)</td>
<td>resume by cases (P)</td>
</tr>
<tr>
<td>(\forall x \ (P(x)))</td>
<td>resume by generalizing (x) from fresh1</td>
</tr>
</tbody>
</table>

If your conjecture looks like \(\forall x \ (P(x))\), then just try whatever method is suggested for \(P(x)\) by itself.

There are some technical caveats for the above suggestions [Garland-Guttag95]. Using resume by \(\land\) can lead to longer proofs, if the same lemma is needed in the proof of each conjunct. Using resume by \(\Rightarrow\) and resume by \(\Leftarrow\) makes certain kinds of proofs by induction impossible (so if you want to do a proof by induction, do that before using these methods). The generalization techniques are not helpful if there are other free variables in the formula.

For complex proofs, it is quite useful to prove lemmas first. You can either state these as things to be proved before what you are trying to prove, or use the prove command during the proof. You may need to use the command set immunity on to keep LP from deleting your lemma after you prove it.

When all else fails, use the old mathematician’s trick of trying to construct a counterexample to your conjecture. If you can’t do it, try to figure out why you can’t (that will suggest a proof technique). Or, if you start making progress towards a counterexample, keep going. Repeat the following mantra to yourself.

\[
\text{It might not be true.}
\]

Finally, when you finally succeed in proving your conjecture, don’t celebrate unless you have also shown that your theory is consistent (see Section 3.17 [How do I prove that my theory is consistent with LP?], page 48).

See Section 3.20 [How can I develop a proof that I can replay later?], page 49, for advice on how to work noninteractively with LP.

For a short, but important list of tips, see the section titled “Sample Proofs: how to guide a proof” in [Garland-Guttag95]. See Chapter 7 of [Guttag-Horning93] for more details.
3.16 What pitfalls are there for LP users?

The biggest pitfall is to not think about what you are doing, but to simply try random proof strategies. This quickly becomes frustrating. See Section 3.15 [Can you give me some tips on proving things with LP?], page 46, for how to work with LP in a more fruitful way.

If your theory is inconsistent, proofs may not be easy, but they will always be possible. Always try to convince yourself of your theory’s consistency before celebrating your proof. See Section 3.17 [How do I prove that my theory is consistent with LP?], page 48.

3.17 How do I prove that my theory is consistent with LP?

It is difficult to prove a theory is consistent using LP. However, one can profitably use LP to search for inconsistency. (See Section 2.18 [What pitfalls are there for LSL specifiers?], page 27, for some common LSL problems that lead to inconsistency.)

One way to use LP to search for inconsistency is using the complete command. This directs LP to attempt to add enough rewrite rules to make the theory equationally complete. If this process finds an inconsistency, such as proving that true == false, then your theory is inconsistent. (This usually doesn’t stop in an acceptable amount of time and space, but if it does, and if it hasn’t found an inconsistency, and if your theory is purely equational, then you have proved your theory is consistent.)

Often the completion procedure seems to go off down some path, generating more and more complex formulas. If this happens, interrupt it (see Section 3.7 [How do I interrupt LP?], page 43) and try using the critical-pairs command to direct the search for inconsistency by naming two sets (not necessarily disjoint) of axioms that you wish to deduce consequences of. You can also use any other of the forward inference techniques in LP (such as the apply command) to try to deduce an inconsistency in a more directed manner.

See [Guttag-Horning93], section 7.6, for more on this topic. See Section 2.21 [Is there a way to write a trait that will guarantee consistency?], page 34, for a discussion of the tradeoffs involved between writing an LSL trait in a way that avoids inconsistency and the difficulty in proving consistency.

3.18 How can I backtrack if I made a mistake in my proof?

If you decide you don’t like the way your current proof is going in LP, use the cancel command to abort the current proof, backtracking to the previous subgoal. Note that this “suspends work on other proofs until an explicit resume command is given” [Garland-Guttag95]. See the documentation for variations of this command.

If you want to start over from scratch, without using the quit command and restarting LP, use the clear command.

3.19 How do I prove something like 0 <= 2 in LP?

As a way of making the problem of proving things like 0 <= 2 concrete, consider the following LSL trait. This trait uses the Integer trait in Guttag and Horning’s handbook (see [Guttag-Horning93], p. 163)
DecimalProblem: trait
  includes Integer
  implies
  equations
  ~(1 <= 0);
  0 <= 2;

The problem is how to coax LP into proving these implications. The following answer is adapted from a posting of Vandevoorde’s ‘comp.specification.larch’ (September 7, 1995).

When the LSL checker translates LSL into LP input, it makes the rewrite rules for the trait DecimalLiterals (see [Guttag-Horning93], p. 164) passive, which means that they don’t get applied automatically [Garland-Guttag95]. For example, 2 will not be rewritten to succ(succ(0)). One reason for this is to keep the conjectures and facts more readable.

However, to prove the implications in your trait, you need to make them active with the following command.

make active DecimalLiterals

This is enough to prove ~(1 <= 0) by rewriting. It’s also a necessary first step for the proofs below.

To prove 0 <= 2, you also need to use facts named IsT0* (from the trait IsT0, see page 194 of [Guttag-Horning93]). The easiest proof to understand is to instantiate the rule for transitivity as follows.

instantiate x by 0, y by 1, z by 2 in IsT0.2

(It’s a good thing you didn’t want to prove 0 <= 9!)

The critical-pairs command is useful for finding such instantiations automatically. For example, to prove 0 <= 2, you can give the following commands to LP.

resume by contradiction % This turns the (negated) conjecture into
% a rewrite rule for critical-pairs.

critical-pairs *Hyp with * % This is common in contradiction proofs.
% For this proof, it suffices to use
% critical-pairs *contraHyp with IsT0.4

3.20 How can I develop a proof that I can replay later?

There are probably many ways to develop a proof that you can replay later. Our technique has the following steps, and starts with an LSL trait. (If you don’t use LSL, you can just use a file of LP input.)

1. Write a trait Foo into the file ‘Foo.lsl’.
2. Use the LSL checker to generate the ‘Foo_Checks.lp’ file.
   % lsl -lp Foo.lsl
3. Optional: use LP to debug your trait and to try out ideas for proofs of the implications interactively.
4. Copy the file ‘Foo_Checks.lp’ into the file ‘Foo.proof’.
   % cp Foo_Checks.lp Foo.proof
5. Optional: if you have used LP already on this trait, then you have a head start. There are lines in the 'Foo.lpscr' that indicate what to do for the proof, and what happened in LP. Copy those lines from the 'Foo.lpscr' and put them in the 'Foo.proof'. For example, after the following line in Foo.proof
   
   prove
   \begin{verbatim}
   (_x1_ \in _x2_) = (_x1_ \in' _x2_)
   \end{verbatim}
   you might want to add the following lines from the script file. The first indicates how you did the proof, and the rest indicate what happened.
   
   resume by induction
   <> basis subgoal
   [] basis subgoal
   <> basis subgoal
   [] basis subgoal
   <> induction subgoal
   [] induction subgoal
   [] conjecture
   
6. Optional: put the command set box-checking on at the top of your proof, so that these box comments will be checked. You only need this once per file.

7. Optional: if you haven’t used LP yet, think about each conjecture, and (if you still believe it), after each conjecture, write a LP command (or commands) that will complete the proof of that conjecture.

8. After each conjecture to be proved, following the commands to prove it, if any, each list of reasons prove command add the LP command qed. For example, your file ‘Foo.proof’, might have a section that looks like the following.
   
   prove
   \begin{verbatim}
   (_x1_ \in _x2_) = (_x1_ \in' _x2_)
   \end{verbatim}
   resume by induction
   <> basis subgoal
   [] basis subgoal
   <> basis subgoal
   [] basis subgoal
   <> induction subgoal
   [] induction subgoal
   [] conjecture
   qed
   
9. Execute the file ‘Foo.proof’ in LP by using the command
   execute Foo.proof
   If this doesn’t work, edit the corrections into the file ‘Foo.proof’. Often you can obtain input for this file from the file ‘Foo.lpscr’.

10. When your proof executes to completion, use the file ‘Foo.lpscr’ to put into ‘Foo.proof’ all of the <> and [] comments that help keep your proof synchronized with LP. When you are done, edit ‘Foo.proof’ so that the differences with the file ‘Foo_Checks.lp’ are minimal, as shown by the following Unix command.
    
    % diff Foo_Checks.lp Foo.proof | more
One advantage of the above procedure is that you can work on one conjecture at a time, which allows you to control the order in which the lemmas are proved. Another advantage is that you tend to think more about what you are doing than if you are interacting directly with LP. Finally, the above procedure is quite helpful when, inevitably, you revise your trait.

When you revise your trait, by editing the file ‘Foo.lsl’, use the following steps to revise your proof.

1. Use the LSL checker to generate the ‘Foo_Checks.lp’ file for the revised trait.
   
   % lsl -lp Foo.lsl

2. Use the Unix `diff` command to check to see what you need to change in your proof.
   
   % diff Foo_Checks.lp Foo.proof | more

   Ignore all the added lines that you put in your proofs such as the following.

   20a21
   > set box-checking on
   78a88,95
   > resume by induction
   > <> basis subgoal
   > [] basis subgoal
   > <> basis subgoal
   > [] basis subgoal
   > <> induction subgoal
   > [] induction subgoal
   > [] conjecture
   > qed

   But pay attention to other differences, particularly those for which there is a `<` at the beginning of the line.

3. Revise your ‘Foo.proof’ file according to the output of the previous step, trying to minimize differences as shown by the `diff` command.

4. Use LP to rerun your proof. In LP, use the command.
   
   execute Foo.proof

   If this doesn’t work, edit the corrections into the file ‘Foo.proof’. Often you can obtain input for this file from the file ‘Foo.lpscr’.

### 3.21 Do I have to reprove everything in an included trait?

No, you should only have to prove that the assumptions from whatever `assumes` clauses you have in the included trait hold in your trait. (See pages 124-125 of [Guttag-Horning93].)

### 3.22 Does LP use assertions in the implies section of an included trait?

According to [Guttag-Horning93] (page 128), LP will reuse the assertions in the `implies` section of a trait without forcing you to prove them again “when it is sound to do so”. [[But when is that exactly?]]
4 Behavioral Interface Specification Languages (BISLs)

In this chapter, we discuss questions about Larch Behavioral Interface Specification Languages (BISLs).

4.1 What is a behavioral interface specification language (BISL)?

A behavioral interface specification language (BISL) is a kind of specification language that has the following characteristics.

- It can specify an exact interface for modules written in some particular programming language. For example, Larch/CLU [Wing83] [Wing87] can specify the details of how a CLU procedure is called, including the types of its arguments, the types of its results, whether it is an iterator, and the names and types of exception results.

- It can specify the behavior of a program module written in that programming language. For example, in Larch/CLU one can specify that a procedure can signal an exception.

An interface describes exactly how to use a module written in some programming language. For example, a function declaration in C tells how to call that function, by giving the function’s name, and by describing the number and types of its arguments.

The concept of behavioral interface specification is an important feature of the Larch family of specification languages [Wing87]. A particularly good explanation of the concept of behavioral interface specification is [Lamport89]. In that paper, Lamport describes a vending machine specification. To specify a vending machine, it is not sufficient to merely describe a mathematical mapping between abstract values (money to cans of soda). One also needs to specify that there will be buttons, a coin slot, a place for the can of soda to appear, and the meaning of the user interface events. The buttons, the coin slot, etc. form the interface of the vending machine; the behavior is described using a mathematical vocabulary that is an abstraction of the physical machinery that will be used to realize the specification. The same two pieces are needed to precisely specify a function to be implemented in, say, C++: the C++ interface information and a description of its behavior.

Many authors use the term interface specification language for this type of language. However, there is a natural confusion between this term and an “interface definition language”, such as the CORBA IDL, which merely specifies interface information and not behavior. To avoid this confusion, we advocate the term “behavioral interface specification language” for any language that specifies not only interface information, but also behavior.

In the Larch family there are several BISLs (see Section 1.1 [What is Larch?], page 3). In the silver book [Guttag-Horning93], a Larch BISL is called a “Larch interface language”. However, we do not use the term “Larch interface language” as a synonym for “BISL”, because there are BISLs outside the Larch family (such as those in the RESOLVE family [Edwards-etal94]). However, “Larch interface language” is a synonym for “Larch BISL”. Another synonym is “Larch interface specification language”.

The first Larch BISL was Larch/CLU [Wing83] [Wing87]. There are now several others. See Section 4.2 [Where can I get a BISL for my favorite programming language?], page 53, for information about how to get an implementation.

See Section 1.1 [What is Larch?], page 3, and [Guttag-Horning93] for more information.
4.2 Where can I get a BISL for my favorite programming language?

See Section 1.1 [What is Larch?], page 3 for what Larch BISLs have been discussed in literature and technical reports. In this section, we concentrate on Larch BISLs that have some kind of tool support. As far as we know, these are the following. (Except as noted, the tools are all free.)

**LCL**  
LCL ([Guttag-Horning93], Chapter 5, [Tan94] [Evans00]) is a BISL for ANSI C. Tool support includes LCLint, which can use specification information to help find errors in programs. The current implementation is available from the following URL.

http://lclint.cs.virginia.edu/

There is also a mailing list for LCLint, which you can subscribe to using the first URL above, or by sending email to lclint-request@virginia.edu.

**Larch/Ada**  
Larch/Ada [Guaspari-Marceau-Polak90] [ORA94] is a BISL for Ada. It is part of the specification environment, Penelope, which also has a proof assistant and a tool, AdaWise, for checking the subset of Ada used in Penelope. AdaWise is free for academic users; for others it can be licensed from Oddessy Research Associates (send email to Peter Rukavena at marketing@oracorp.com). Information on Penelope and AdaWise is available at the following URLs.

http://www.oracorp.com/ada/penelope.htm  
http://www.oracorp.com/ada/adawise.htm

**Larch/Smalltalk**  
Larch/Smalltalk [Cheon-Leavens94] [Cheon-Leavens94b] is a BISL for Smalltalk-80. It has a syntax and type checker, as well as a specification browser. It runs under ParcPlace Objectworks\Smalltalk. It can be obtained from the following URLs.

http://www.cs.iastate.edu/~leavens/larchSmalltalk.html  
ftp://ftp.cs.iastate.edu/pub/larchSmalltalk/

**Larch/C++**  
Larch/C++ [Leavens97] [Leavens96b] is a BISL for C++. The current release has a syntax checker, examples, and a reference manual. It can be obtained from the following URLs.

http://www.cs.iastate.edu/~leavens/larchc++.html  

**VSPEC**  
VSPEC [Baraona-Penix-Alexander95] [Baraona-Alexander96] [Baraona-Alexander-Wilsey96] is a BISL for VHDL. The current release has a syntax checker. It can be obtained from the following URLs.

http://www.ece.uc.edu/~pbaraona/vspec/index.html  
ftp://ftp.ece.uc.edu/pub/users/pbaraona/vspec2.0.tar.gz

**GCIL**  
The Generic Concurrent Interface Language, GCIL, [Lerner91] has a syntax and type checker. It can be obtained from the following URL.

4.3 Do I need to write an LSL trait to specify something in a BISL?

No, you don't always have to write an LSL trait to specify something in a BISL, it just seems that way. Seriously, you don't have to write a trait in the following circumstances.

- You are specifying a procedure and all of the mathematical vocabulary needed is available in existing traits. These traits may come from a handbook (see Section 2.23 [Where can I find handbooks of LSL traits?], page 37), or they may be built-in to your BISL. For example, in most Larch BISLs, you would not have to write a trait for your language's built-in integer type.
- Your BISL may create a trait for some compound type for you automatically. For example, in LCL, traits are created automatically to model C `struct` types.
- You have imported a specification file that defines an abstract data type (ADT), which is based on an already written trait.

If you are specifying an ADT in a Larch BISL, you will always have to use some trait to describe its abstract values (see Section 4.4 [What is an abstract value?], page 55). It might be a trait from a handbook, or some combination of traits that already exist, but there will have to be one that defines the abstract values for your ADT.

Note that it is almost always possible to use one of LSL's shorthands or a combination of handbook traits to specify the abstract values of an ADT. The only drawback might be that such descriptions of abstract values might not be as close to your way of thinking about your problem domain as you might like or it might not have enough high-level vocabulary for your purposes. For example, when specifying the abstract values of symbol tables, one might use the handbook trait `FiniteMap` (see [Guttag-Horning93], page 185). However, if your symbol table has to handle nested scopes, that trait, by itself might not be adequate. You might, for example, want to use a stack of finite maps as abstract values, in which case you might also use the both the `FiniteMap` and `Stack` traits (see [Guttag-Horning93], page 170), and also add some additional operators to search the abstract values appropriately. If that kind of thing gets too complex, then it might be better to start writing your own trait (see Section 2.20 [How do I know when my trait is abstract enough?], page 33). Also, when using LSL shorthands and standard handbook traits, be careful that you do not state contradictory constraints as axioms (see Section 2.18 [What pitfalls are there for LSL specifiers?], page 27).

Usually, when you are specifying a procedure, you won't have to specify a separate trait, because you will use the vocabulary of the traits that describe the abstract values of whatever types of arguments and results (and global variables) your procedure uses. However, if this vocabulary isn’t expressive enough, then you may find it necessary or convenient (see [Wing95], Section 4.1) to write your own trait. This trait would specify the additional LSL operators you need to state your procedure's pre- and postconditions; it would normally assume the traits describing the abstract values of the procedure’s argument and result types (see Section 2.8 [What is the difference between assumes and includes?], page 17).

The way you tell your BISL what trait it should use differs between BISLs. For example, in LCL and Larch/C++, the keyword that starts the list of traits to be used is `uses`. In Larch/ML, the keyword is `using`; in Larch/CLU, it is `based on`; in LM3 it is `TRAITS`; in Larch/Smalltalk it is `trait`, in Larch/Ada it is `TRAIT`. 
4.4 What is an abstract value?

An abstract value is a mathematical abstraction of some bits that might exist in your computer. (It’s an abstraction in the same sense that a road map of the United States is an abstraction of the geography and roads of the United States.) In a Larch BISL’s model of your computer, each variable contains an abstract value. The idea is that, you shouldn’t have to reason about the bit patterns inside the computer, which are mathematically messy, far removed from the concepts that are important in your problem domain, and subject to change. Instead, you should reason about your programs using simple ideas that are close to your problem and independent of the changing details of your program’s data structures.

This idea originated in Hoare’s article on specification and verification of ADTs [Hoare72a]. Hoare’s example used an ADT of small integer sets, which were implemented using arrays, but specified using mathematical sets. The mathematical sets were thus the abstract values of the arrays used in the implementation.


4.5 What do mutable and immutable mean?

An object is mutable if its abstract value can change over time. (See Section 4.4 [What is an abstract value?], page 55 for more about abstract values.) For example, most collection types, such as arrays or records in typical programming languages are types with mutable objects.

By contrast, an object is immutable if its abstract value cannot change. For example, the integers are not mutable, as you cannot change the number 3 to be 4.

Typically, immutable objects are atomic and do not contain subobjects, while mutable objects often contain subobjects.

A type with mutable objects is sometimes called an mutable type and a type with immutable objects is called an immutable type. The specifications of mutable and immutable types differ in that an immutable type cannot have operations that mutate objects of that type.

4.6 If I specify an ADT in a BISL do I prove it is the same as the trait?

No, you don’t prove the specification of an ADT in a BISL is the same as a trait. Those are completely different concepts. An LSL trait specifies a theory, not what is to be implemented in a programming language; the BISL is what is used to specify an implementation (program module). You use the vocabulary in a trait to help specify program modules in a BISL, but there is no proof obligation that you engender by doing so. See Section 1.1 [What is Larch?], page 3, and Chapter 3 of [Guttag-Horning93], for more on the general idea of two-tiered specifications.

What you might want to prove instead is the correctness of an implementation of your ADT specification; for example, if you specify a class Stack in the BISL Larch/C++, then you would use an LSL trait to aid the specification (see Section 1.2 [Why does Larch have two tiers?], page 4), and implement the class in C++. You could then try to prove that
your C++ code is correct with respect to your Larch/C++ specification. In this proof, you could make use of the trait’s theory. However, you wouldn’t be proving any correspondence between the trait and the BISL specification.

4.7 What does requires mean?

In most Larch BISLs\(^1\), the keyword requires introduces a precondition into a procedure specification. (This usage dates back to at least Larch/CLU [Wing83].)

A precondition is a predicate that must hold before a the procedure is invoked. It documents the obligations of the caller in the contract that is being specified. Therefore, it can be assumed to be true in an implementation of the procedure, because if it is does not hold when the procedure is not obligated to fulfill the contract. This semantics is a “disclaimer” semantics [Jackson95] [Khosla-Maibaum87]; it differs from an “enabling condition” semantics, which would mean that the procedure would not run if called in a state that did not satisfy the precondition.

As an example, consider the specification of a square root function, which is to be written in C (see [Guttag-Horning93], page 2). In this specification the precondition is specified by the following line.

\[
\text{requires } x \geq 0;
\]

This means that if the value of the formal parameter \( x \) is not greater than or equal to 0, then the specification says nothing about what the implementation should do. Thus that case does not have to be considered when writing an implementation.

If the precondition of a procedure is the constant predicate true, the entire requires clause can usually be omitted. Such procedures can always be called, and an implementation can make no assumptions about the call except that the formal parameters will all have whatever types are specified. Because such a procedure will have to validate any properties it needs from its arguments, it may be safer in the sense of defensive programming. However, it may also be slower than one that requires a stronger precondition, because it will have to spend time validating its arguments. This is a design tradeoff that can be made either way, but documenting preconditions in procedure specifications prevents the common performance problem of having both the caller and the procedure checking the validity of the arguments. (See [Liskov-Guttag86] for more discussion of this issue.)

A precondition refers to the state just before the specified procedure’s body is run (but after parameter passing). The term comes from Hoare’s original paper on program verification [Hoare69], in which each piece of program code was described with two predicates: a precondition and a postcondition. Both the pre- and the postcondition were predicates on states (that is, they can be considered functions that take a state and return a boolean). The precondition described the pre-state; that is, it describes the values of program variables just before the execution of a piece of code. The postcondition described the post-state, which is the state after the code’s execution. In a Larch BISL the precondition is like this,

\(^1\) The semantics of a Larch BISL is not fixed by being a member of the family, but there is a family resemblance, due mostly to the common heritage from Larch/CLU [Wing83]. The remarks about BISLs should be taken as generalities about the family, especially about BISLs for sequential programming languages like C and C++, and might not hold for your particular BISL. Consult the documentation for your particular BISL to be sure.
but the postcondition is typically a predicate on both the pre- and the post-state. See Section 4.8 [What does ensures mean?], page 57, for more information.

4.8 What does ensures mean?

In most Larch BISLs, the keyword ensures introduces a postcondition into a procedure specification. (This usage dates back to at least Larch/CLU [Wing83].)

A postcondition in a Larch BISL is typically a predicate that is defined on the pre-state and post-state of a procedure call. (See Section 4.7 [What does requires mean?], page 56, for background and terminology.) It documents the obligations of the called procedure in the contract that is being specified. Therefore, it can be assumed to be true at the end of a call to the procedure, when reasoning about the call.

As an example, consider the following specification of an integer swap function which is to be written in C++ (this example is quoted from [Leavens97]).

```cpp
extern void swap(int& x, int& y) throw();
//@ behavior {
//@ requires assigned(x, pre) /
//@ assigned(y, pre);
//@ modifies x, y;
//@ ensures x' = y^ /
//@ y' = x^;
//@ }
```

In this specification, the parameters x and y are passed by reference, and thus each is an alias for a variable in the caller’s environment. Because x and y denote variables, they can have different values in the pre-state and the post-state. The postcondition of the example above says that the pre- and post-state values of the variables x and y are exchanged. The notation x^ refers to the value of x in the pre-state, and x' refers to its value in the post-state. (This Larch/C++ notation is taken from LCL. In LM3 the ^ notation is not used. In Larch/Smalltalk, the ^ is likewise dropped, and a subscript “post” is used instead of ‘.)

Depending on the particular BISL, a procedure specification may have one of two different interpretations. In most Larch BISLs for sequential programming languages, the interpretation is that if the precondition holds, then the procedure must terminate normally in a state where the postcondition holds. This is called a total correctness interpretation of a procedure specification. In Hoare’s original paper [Hoare69], and in GCIL and Larch/CORBA, a partial correctness interpretation is used. This means that if the precondition holds and if the procedure terminates normally, then the postcondition must hold. You should check the documentation for your BISL to be sure of its semantics on this point. (Larch/C++ is unique in that it allows you to specify either interpretation; see [Leavens97], Chapter 6).

4.9 What does modifies mean?

In most Larch BISLs, the keyword modifies introduces a list of objects that can be modified by an execution of the specified procedure. (This usage dates back to Larch/CLU [Wing83].)

An object is modified if its abstract value is changed by the execution of the specified procedure. (That is, if it has a different abstract value in the pre-state from the post-state.) Note that the modifies clause does not mean that the execution must change the
object’s abstract values; it simply gives the procedure permission to change them. For example, if a `swap` procedure were passed variables with identical values (see Section 4.8 [What does ensures mean?], page 57, for the specification of `swap`), then the values would not be changed.

Note also that a variable’s bits may change without a corresponding change in abstract value (see Section 4.4 [What is an abstract value?], page 55). For example, if one has a rational number object, whose representation uses two integers. Let us write the two integers in the representation as \((i, j)\); thus \((2, 4)\) might be one bit pattern that represents the abstract value \(1/2\). However, changing the bits from \((2, 4)\) to \((1, 2)\) would not affect the abstract value.

The `modifies` clause can be considered syntactic sugar for writing a larger predicate in the postcondition. Since only the objects listed in the `modifies` clause can have their abstract values changed by the execution of the procedure, all other objects must retain their abstract values. There is a problem in artificial intelligence called the “frame problem”, which is how to know that “nothing else has changed” when you make some changes to the world. For this reason, the `modifies` clause in a Larch BISL can be said to specify a frame axiom for the specified procedure. See [Borgida-etal95] for a general discussion of the frame problem for procedure specifications.

In many specification languages, the frame axiom for a procedure specification is stated in terms of variable names, but in Larch BISLs it is typically stated in terms of objects. The difference is crucial when the programming language allows aliasing. If the `modifies` clause in a Larch BISL specified that only certain variable names could change their values, then one would typically need to have either prohibit aliasing completely, or one would need to write preconditions that prohibited aliases to variables that were to be modified in procedures.

Mentioning an object in a `modifies` clause should not give the procedure being specified permission to deallocate the object. If that were allowed, it would lead to problems in the BISL’s semantics [Chalin95] [Chalin-etal96]. See Section 4.10 [What does trashes mean?], page 58, for more information on this topic.

### 4.10 What does trashes mean?

A `trashes` clause is similar to a `modifies` clause (see Section 4.9 [What does modifies mean?], page 57) in a procedure specification. It gives the procedure permission to deallocate or trash the objects listed. These objects do not have to be deallocated or trashed, but no other objects may be deallocated or trashed.

The notion of `trashing` an object means that its abstract value should not be used again. In Larch/C++, allocated variables can be in one of two states: assigned or unassigned; a variable is unassigned if it has not been assigned a sensible value. However, if one assigns to an assigned variable, \(a\), the value of an unassigned variable, then \(a\) becomes unassigned, and hence is considered to be trashed.

Chalin [Chalin95] [Chalin-etal96] pointed out that specification of what a procedure can modify should be separated from the specification of what it can deallocate or trash. When these notions are separated, as they are in Larch/C++ (see [Leavens97], chapter 6), there are two frame axioms for procedure specifications.
An alternative to a *trashes* clause, used in LCL, is to specify input/output modes for formal parameters [Evans00]. Some parameter modes do not allow trashing, and some do not allow modification.

### 4.11 What does claims mean?

The *claims* clause, and other kinds of checkable redundancy for BISLs, were introduced into LCL by Tan [Tan94]. In a procedure specification, the keyword *claims* introduces a predicate, called a claim, that, like the postcondition, relates the pre- and post-states of a procedure’s execution. Roughly speaking, the meaning is that the conjunction of the precondition and the post condition should imply the claim. This implication can be checked, for example, by using LP.

You could use the *claims* clause if you wish to highlight some consequences of your procedure specification for the reader. It can also be used to help debug your understanding of a procedure specification; for example, if you think of more than one way to express a postcondition, write all but one way in a *claims* clause. Thus the purpose of this clause is similar to the purpose of the *implies* section of an LSL trait (see Section 2.11 [When should I use an implies section?], page 20).

### 4.12 What is the meaning of a specification written in a BISL?

The exact meaning of a specification in a BISL is, of course, described by the reference manual for that particular BISL. But in general, the meaning of a BISL specification is a set of program modules that satisfy the specification. These program modules are to be written in the programming language specified. For example, the meaning of a LCL specification is a set of ANSI C program modules that satisfy the specification. Similarly, the meaning of a Larch/C++ specification is a set of C++ program modules that satisfy the specification; a Larch/C++ specification cannot be implemented in, say, Ada or Modula-3.

How does one describe the satisfaction relation for a BISL? Suppose you want to describe satisfaction for the BISL Larch/X, whose programs are to be written in language X. Then you describe the satisfaction relation in at least the following ways.

- Give a semantics for language X, that assigns to each program module, P, a meaning, M(P). Describe, for each specification, S, written in Larch/X, when M(P) is in the set of meanings specified by S.
- Develop a verification logic for language X, that says how to check that code satisfies a specification (or some other specification). Then a program module P satisfies a specification S if one can use the verification logic to check that P satisfies S.

The first technique will probably allow more programs, in principle, to satisfy the specification, especially if the semantics of language X covers all aspects of the language. The second technique is probably more limiting, but is certainly more useful in practice. These techniques are not mutually exclusive; for example, a relatively complete verification logic could be used as a semantics for that language. Furthermore, by using complimentary definitions of the programming language (e.g., a denotational semantics and a verification logic), one could prove soundness and completeness results, which would help debug each kind of semantics, and thus the semantics of the BISL as well.
See [Wing83] for a semantics of Larch/CLU. See [Jones93] for a semantics of LM3. See [Tan94] [Chalin95] for semantics of LCL.

4.13 How do I specify something that is finite or partial?

In a procedure specification, you would normally use a precondition to limit the acceptable arguments that the procedure can work with. For example, the standard LSL handbook trait `Integer` (see [Guttag-Horning93], page 163) specifies an unbounded set of integers. Many Larch BISLs use this kind of unbounded set of integers as their model of a programming language’s built-in integer type. Suppose that you want to specify an integer `discriminant` function, say to be implemented in C++. In Larch/C++ you could specify this as follows.

```c++
extern int discriminant (int a, int b, int c) throw();
//@ behavior {
//@ requires inRange((b * b) - (4 * a * c));
//@ ensures result = (b * b) - (4 * a * c);
//@ }
```

Notice the precondition; it says that the result must be able to be represented as an `int` in the computer (according to the definition of the operator `inRange`, which is defined in a built-in trait of Larch/C++ [Leavens97]). Without this precondition, the specification could not be implemented, because if `b` were the maximum representable integer, and if `a` and `c` were 0, then the specification without the precondition would say that the execution would have to terminate normally and return an integer larger than the maximum representable integer.

(The specification of `discriminant` above may not be the best design for the procedure, because it requires a precondition that is difficult to test without performing the computation specified. See Section 4.17 [Can you give me some tips for specifying things in a BISL?], page 63, for more discussion of this point.)

The idea of using preconditions to get around partiality and finiteness problems in procedures goes back to Hoare’s original paper on program verification [Hoare69].

To specify an ADT with a finite set of values, such as a bounded queue, use a trait with unbounded values to specify the abstract values. You can then use an invariant in the BISL specification of your ADT to state that the size of the abstract values is limited. Most likely the ADT operations that construct and mutate objects would need to have preconditions that prevent the abstract value from growing too large. This approach is preferred over an approach that specifies a finite set of abstract values in LSL, because that trait would be more complicated and less likely to be able to be reused (see Section 2.20 [How do I know when my trait is abstract enough?], page 33). Even with such a trait, you would still need to have preconditions on your operations that construct and mutate objects, so there is little gained.

4.14 How do I specify errors or error-checking?

Ways of specifying errors and error checking vary between Larch BISLs, depending primarily on whether the programming language that the BISL is tailored to has exceptions. In a language without exceptions (like ANSI C), a procedure would have to signal an error by
producing an error message and halting the program, or by returning some error indication to the caller of a procedure. If you want to specify that the program halts, most likely you will just not include the conditions under which it can do so in your precondition (see Section 4.13 [How do I specify something that is finite or partial?], page 60). If you want to specify the returning of some error indication (a return code, etc.), this is done as if you were specifying any other normal result of a procedure. In a language with exceptions (like C++), there is the additional option of using the exception handling mechanism to signal errors to the caller. The syntax of this varies, but typically you will declare the exceptions as usual for the programming language, and there will be some special way to indicate in the postcondition that an exception is to be raised.

Many Larch BISLs (including LCL, LM3, Larch/Ada, and Larch/C++) have special syntax to support the specification of error checking and signalling exceptions. However, there is little common agreement among the various Larch BISLs on the syntax for doing so; please check the manual for your particular BISL. (See Section 1.1 [What is Larch?], page 3, for how to find your language manual.)

See [Liskov-Guttag86] for a discussion of the tradeoffs between using preconditions and various kinds of defensive programming.

4.15 How do I specify that all the values of a type satisfy some property?

A property that is true for all values of a type is called an invariant. Many Larch BISLs have syntax for declaring invariants. For example, LM3 uses the keyword \texttt{TYPE_INVARIANT}, and Larch/C++ uses the keyword \texttt{invariant}. The predicate following the appropriate keyword has to hold for all objects having that type, whenever a client-visible operation of that type is called or returns. (This allows the invariant to be violated temporarily within an operation implementation.)

Stating an invariant in the BISL specification of an ADT is often preferable to designing an LSL trait so that the abstract values all satisfy the property. For example, if you attempt to state such a property as an axiom in a trait, you can cause the trait to be inconsistent, if the abstract values of your type are freely generated (see Section 2.18 [What pitfalls are there for LSL specifiers?], page 27). But invariants in a BISL do not lead to such inconsistencies, because they describe properties of objects created by an ADT, not properties of all abstract values.

See Section 4.13 [How do I specify something that is finite or partial?], page 60, for more discussion about ways to specify that all values of an ADT are finite.

4.16 What pitfalls are there for BISL specifiers?

The main thing to watch out for in a BISL specification is that you don’t specify something that cannot be implemented. It’s worth repeating the following mantra to yourself whenever you write a procedure specification.

| Could I correctly implement that? |
Of course, you would never intentionally specify a procedure that cannot be implemented. So you are only likely to fall into this pit by accident. The following are some ways it might happen.

A very simple problem is forgetting to write a \texttt{modifies} (or \texttt{trashes}) clause in a procedure specification when your postcondition calls for some object to change state (or be deallocated). Remember that leaving out a \texttt{modifies} clause means that no objects can be modified. Consider the following bad specification of an increment procedure in Larch/C++. (The formal parameter \texttt{i} is passed by reference. The \texttt{throw()} says that no exceptions can be thrown. In the precondition, the term \texttt{assigned(i, pre)} is a technical detail; it says that \texttt{i} has been previously assigned a value [Leavens97].)

\begin{verbatim}
extern void incBad(int& i) throw();
//@ behavior {
//@ requires assigned(i, pre);
//@ ensures i' = i^ + 1;
//@ }
\end{verbatim}

There can be no correct implementation of the above specification, because the omitted \texttt{modifies} clause does not allow \texttt{i} to be modified, but the postcondition guarantees that it will be modified. This contradiction prevents correct implementations.

A more subtle way that your specification might not be implementable is if you forget to specify a necessary precondition. Suppose we fix the above specification and produce the following specification of an increment function. What's wrong now?

\begin{verbatim}
extern void incStillBad(int& i) throw();
//@ behavior {
//@ requires assigned(i, pre);
//@ modifies i;
//@ ensures i' = i^ + 1;
//@ }
\end{verbatim}

This specification can't be implemented, because if the pre-state value of \texttt{i} is the largest \texttt{int}, \texttt{INT_MAX}, then the value \texttt{INT_MAX+1} can't be stored in it. So, as specified, there are no correct implementations. Something has to be done to accommodate the finite range of integers that can be stored in \texttt{i}. Changing the precondition to be the following would take care of the problem.

\texttt{assigned(i, pre) /\ i^- < INT_MAX}

In summary, one thing to watch for is whether you have taken finiteness into account. See Section 4.13 [How do I specify something that is finite or partial?], page 60, for more about finiteness considerations in BISL specifications.

Another way that your specification might be unimplementable is if your stated post-condition contradicts some invariant property (see Section 4.15 [How do I specify that all the values of a type satisfy some property?], page 61) that you specified elsewhere.

The following is another Larch/C++ example. The term \texttt{inRange(e)}, used in the precondition, means that \texttt{e} is within the range of representable integer values, hence the specification takes finiteness into account. What else could be wrong with the specification?
extern void transfer(int& source, int& sink, int amt);
//@ behavior {
//@ requires assigned(source, pre) \& assigned(sink, pre) \& amt >= 0
//@ \& inRange(source^ - amt) \& inRange(sink^ + amt);
//@ modifies source, sink;
//@ ensures source' = source^ - amt \& sink' = sink^ + amt;
//@ }

The above specification can't be implemented because it doesn't require that source and sink not be aliased. Since both source and sink are passed by reference, suppose that both refer to the same variable i. Then if amt is nonzero, the postcondition cannot be satisfied. (Suppose the pre-state value of i is 10, and that amt is 5, then the post-state must satisfy source' = 10 - 5 \& sink' = 10 + 5, but since both source and sink are aliases for i, this would require that i' be both 5 and 15, which is impossible.) In Larch/C++, this could be fixed by adding the following conjunct to the precondition, which requires that source and sink not be aliased.

source ~= sink

Some Larch BISLs (e.g., Larch/Ada) prohibit aliasing in client programs, so if you are using such a BISL, then you don't have to worry about this particular way of falling in the pit. But if your BISL does not prohibit aliasing, then you should think about aliasing whenever you are dealing with mutable objects or call-by-reference.

4.17 Can you give me some tips for specifying things in a BISL?

A basic tip for writing a BISL specification is to try to permit as many implementations as possible. That is, you want to specify a contract between clients and implementors that does everything the client needs, but does not impose any unneeded restrictions on the implementation.

One thing that experienced specifiers think about is whether the specification permits implementations to be tested. This is particularly important for preconditions. For example, when specifying an ADT, check that the ADT has enough operations to allow the pre- and postconditions specified to be tested. If you specify a Stack ADT but leave out a test for the empty stack, then it will be impossible for a client to test the precondition of the pop operation. (However, there are some rare examples, such as a statistical database, where you might not want to allow clients to have such access.)

If implementations can be tested, it is often useful to compare the difficulty in testing the specified precondition with the difficulty of doing the specified computation. For example, in the specification of discriminant (see Section 4.13 [How do I specify something that is finite or partial?], page 60), checking the precondition is likely to be as difficult as the specified computation. In such a case, it may be wise to change the specification by weakening the precondition and allowing the procedure to signal some kind of error or exception.

Another tip is to try to use names for operators in traits that are distinct from the name of the procedure you are trying to specify (or vice versa). For example, the casual reader might incorrectly think that the following specification is recursive.
/@ uses FooTrait;

extern int foo(int input);
//@ behavior {
//@   ensures result = foo(input);
//@ }

There is no logical problem with the above example, since the operators used in a postcondition cannot have anything to do with the interface procedure being specified. However, naive readers of specifications are (in our experience) sometimes confused by such details.

See Section 4.16 [What pitfalls are there for BISL specifiers?], page 61, for pitfalls to avoid. See [Liskov-Guttag86] [Meyer97] [Meyer92] for general advice on designing and specifying software. See [Wing95] for some other tips on formal specification.
Bibliography

Please send corrections in this bibliography to larch-faq@cs-DOT-iastate-DOT-edu.

[Baraona-Alexander-Wilsey96]

[Baraona-Alexander96]

[Baraona-Penix-Alexander95]

[Baumeister96]

[Borgida-etal95]

[Bowen-Hinchey95]

[Bowen-Hinchey95b]

[Boyer-Moore79]

[Boyer-Moore88]


[Goguen84]

[Goguen86]

[Gordon-Melham93]

[Gries-Schneider95]

[Guaspari-Marceau-Polak90]

[Guttag-Horning-Modet90]

[Guttag-Horning-Wing85a]

[Guttag-Horning-Wing85b]

[Guttag-Horning78]

[Guttag-Horning91b]

[Guttag-Horning93]
and Monographs in Computer Science series (Springer-Verlag, NY, 1993). (The ISBN numbers are 0-387-94006-5 and 3-540-94006-5.) This is currently out of print, but a (large) postscript file for the entire book can be obtained from the following URL

'http://www.sds.lcs.mit.edu/spd/larch/pub/larchBook.ps'.

Appendix A is available separately from the following URLs.

'http://www.research.digital.com/SRC/larch/toc.html'

'http://www.research.digital.com/SRC/larch/handbook.ps'

### [Guttag75]

### [Hall90]

### [Hayes93]

### [Hoare69]

### [Hoare72a]

### [ISO-VDM96]

### [Jackson95]

### [Jones90]

'http://www.cs.ncl.ac.uk/people/cliff.jones/home.formal/research-other.html'.

### [Jones91]


### [Jones93]
Kevin D. Jones. A Semantics for a Larch/Modula-3 Interface Language. In [Martin-Wing93], pages 142-158.

### [Jonkers91]

[Kaufmann-Moore97]

[Khosla-Maibaum87]

[Lamport89]

[Leavens97]

[Leavens96b]

[Leavens-Wing97]


[Loeckx-Ehrich-Wolf96]

[Luchangco-etal94]

[Martin-Lai92]

[Martin-Wing93]

[Meyer92]

[Meyer97]

[Mosses96]

[ORA94]

[Owre-etal95]

[Paulson94]

[Saiedian-etal96]
[Saxe-etal92]

[Schmidt86]

[Sitaraman-Welch-Harms93]

[Sivaprasad95]

[Soegaard-Anderson-etal93]


[Staunstrup-Garland-Guttag89]

[Staunstrup-Garland-Guttag92]


[Wing-Rollins-Zaremski93]
Jeannette M. Wing, Eugene Rollins, and Amy Moormann Zaremski. Thoughts on a Larch/ML and a New Application for LP. In [Martin-Wing93], pages 297-312.


Example Index

A
AnswerTrait ......................................... 16
AnswerTrait2 ....................................... 19

B
BadDefinitionsTrait ................................. 36
BadRational ........................................... 29
BadRecTrait ........................................... 26
BadSig ................................................... 29
BadSubsetTrait ........................................ 30
BadZeroTrait ......................................... 26

C
ClockTrait .............................................. 30
ConvertsExample .................................... 21

D
DecimalProblem ....................................... 48
DenseOrder ............................................. 38
discriminant ........................................... 60

F
foo ....................................................... 63

G
GuasparisExample .................................... 27
GuasparisExampleCorrected ........................ 27

I
incBad .................................................... 62
incStillBad .......................................... 62

L
LSL_rational .......................................... 8
LSL_rational_sort .................................... 8

M
Makefile, for LSL trait checking .................. 15

N
NonAbstractSet ....................................... 33

P
PhoneBookTrait ....................................... 7

Q
QTrait ................................................... 24

R
ResponseSigTrait .................................... 31
ResponseTrait ......................................... 31
ResponseTraitExercise .............................. 32

S
swap .................................................... 57

T
transferBad ......................................... 62

Z
Z_rational .............................................. 8
Concept Index

Please send corrections in this index, including entries that you found that were not helpful, and things you found that weren’t indexed, to larch-faq@cs-DOT-iastate-DOT-edu.

- 1, writing in LSL ........................................... 33
-lp switch, on LSL checker ................................. 44

\n
\A, in LSL ..................................................... 38
\E, in LSL ..................................................... 38
\forall, not within an LSL term ............................. 38

A
abstract value ................................................. 55
abstract value, example of ................................... 55
abstract value, vs. bits in modifies clause ............... 58
abstraction ..................................................... 12
abstractness, of LSL trait ................................... 33
acknowledgements ............................................ 2
activating, literals in LP .................................... 49
Ada, Larch BISL for ........................................... 3, 53
ADT, BISL specification and traits .......................... 54
ADT, correctness of vs. trait .................................. 55
ADT, i.e., an abstract data type ............................ 54
ADTs, specification of with finite values ................. 60
Alexander ...................................................... 2
algebraic model, of a trait ................................. 23
aliasing ......................................................... 4
aliasing, and the modifies clause .......................... 58
aliasing, in a BISL specification ......................... 62
ANSI C, Larch BISL for ...................................... 3
ASCII codes for LSL symbols ............................... 38
assertions in LSL, tip for making them abstract ......... 32

asserts section, position of ................................ 17
assumes clause, position of ................................ 17
assumes clause, proof obligations in LP ................... 51
assumes vs. includes ......................................... 17
assumption, about calls, specifying in a BISL ......... 56
automatic proof techniques, in LP ......................... 45
axioms ......................................................... 20
Axioms file .................................................. 44

B
backspace, in LP .............................................. 42
backtracking, in a LP proof ................................. 48
Baker ........................................................... 2
based on in Larch/CLU ....................................... 54
bearings, getting them in LP ............................... 45
behavior ....................................................... 4
behavioral interface specification language ............. 3, 52
Bekaaert ....................................................... 2
bibliographic references ..................................... 1
BISL ........................................................... 3
BISL semantics ................................................ 59
BISL specification of errors .................................. 60
BISL specification, pitfalls ................................. 61
BISL specification, tips ....................................... 63
BISL, defined ................................................. 52
BISL, differences from LSL ................................. 5
BISL, for object-oriented languages ........................ 12
BISL, invariant for a type .................................... 61
BISLs, getting one for language X .......................... 53
bits, vs. abstract value in modifies clause ............... 58
book, silver ................................................... 11
books, on Larch ................................................ 11
Borgida .......................................................... 58
bottom ........................................................... 25
bottom, not in LSL ............................................ 24
Bowen ........................................................... 12
Boyer ........................................................... 35

C
C, Larch BISL for .............................................. 3, 53
C++, Larch BISL for ......................................... 3, 53
cancel command in LP ....................................... 48
cases, proof by ............................................... 19
checkable redundancy in a BISL ......................... 59
checking consistency with LP .............................. 48
checking LSL traits, makefile for ......................... 15
checklist, for abstraction ................................... 33
Checks file .................................................... 44
checks, of a trait with LP ................................... 43
Cheng ........................................................... 15
claim, in LSL, a consequence .............................. 20
claims clause .................................................. 59
claims, of unique-definition ............................... 21
clear command in LP ........................................ 48
CLU, Larch BISL for .......................................... 3
CLU, used for LP’s implementation ....................... 42
COLD-K ........................................................ 9
COLD-K vs. Larch BISLs ..................................... 9
COLD-K vs. LSL ............................................... 9
commands in LP, summary ................................... 42
comp.specification.larch, newsgroup ....................... 11
complete command in LP, use in searching for ....... 48
inconsistency .................................................. 48
<table>
<thead>
<tr>
<th>Concept</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>complete specification, not necessary in LSL</td>
<td>25</td>
</tr>
<tr>
<td>concurrency, Larch BISL for</td>
<td>3</td>
</tr>
<tr>
<td>concurrent programming, Larch BISL for</td>
<td>53</td>
</tr>
<tr>
<td>consequences, specifying</td>
<td>20</td>
</tr>
<tr>
<td>consistency, ensuring in LSL</td>
<td>34</td>
</tr>
<tr>
<td>consistency, proving it with LP</td>
<td>48</td>
</tr>
<tr>
<td>consistency, vs. ease of expression</td>
<td>35</td>
</tr>
<tr>
<td>constraining a type</td>
<td>9</td>
</tr>
<tr>
<td>constraining constants</td>
<td>9</td>
</tr>
<tr>
<td>constraint, in Z</td>
<td>9</td>
</tr>
<tr>
<td>constructor, a generator in LSL</td>
<td>31</td>
</tr>
<tr>
<td>context, section of a trait</td>
<td>17</td>
</tr>
<tr>
<td>contract</td>
<td>12</td>
</tr>
<tr>
<td>conversion</td>
<td>21</td>
</tr>
<tr>
<td>conversion from LSL to HTML</td>
<td>39</td>
</tr>
<tr>
<td>converts clause</td>
<td>21</td>
</tr>
<tr>
<td>converts clause, use it!</td>
<td>20</td>
</tr>
<tr>
<td>converts, use of</td>
<td>21</td>
</tr>
<tr>
<td>CORBA-IDL, Larch BISL for</td>
<td>3</td>
</tr>
<tr>
<td>correctness, partial vs. total</td>
<td>57</td>
</tr>
<tr>
<td>correctness, with LP</td>
<td>45</td>
</tr>
<tr>
<td>counterexamples, search for</td>
<td>47</td>
</tr>
<tr>
<td>critical-pairs command in LP, use in searching for inconsistency</td>
<td>48</td>
</tr>
<tr>
<td>decimals</td>
<td>13</td>
</tr>
<tr>
<td>decimals, using it to run LP</td>
<td>42</td>
</tr>
<tr>
<td>decimals, using with LP</td>
<td>15</td>
</tr>
<tr>
<td>enabling condition (guard)</td>
<td>9</td>
</tr>
<tr>
<td>ensures clause, in a BISL</td>
<td>57</td>
</tr>
<tr>
<td>enumeration clause, position of</td>
<td>17</td>
</tr>
<tr>
<td>equation, basis of LSL</td>
<td>14</td>
</tr>
<tr>
<td>equations, what to write</td>
<td>31</td>
</tr>
<tr>
<td>equivalence class</td>
<td>27</td>
</tr>
<tr>
<td>error character, in LP</td>
<td>42</td>
</tr>
<tr>
<td>error checking, specifying them</td>
<td>60</td>
</tr>
<tr>
<td>errors, specifying them</td>
<td>60</td>
</tr>
<tr>
<td>examples, in a trait</td>
<td>20</td>
</tr>
<tr>
<td>exceptions, to conversion claims</td>
<td>22</td>
</tr>
<tr>
<td>exempting and undefined terms</td>
<td>22</td>
</tr>
<tr>
<td>exempting clause</td>
<td>22</td>
</tr>
<tr>
<td>exempting clause, with conditions</td>
<td>23</td>
</tr>
<tr>
<td>existential quantifiers, in LSL terms</td>
<td>38</td>
</tr>
<tr>
<td>extensions</td>
<td>32</td>
</tr>
</tbody>
</table>

**D**
- debugging | 13
- debugging a BISL specification | 59
- debugging, a trait | 20
- debugging, a use of LP | 40
- decimal literals | 48
- defensive programming, vs. preconditions | 56
- definition principle | 35
- DeMillo | 12
- Dhara | 2
- disadvantages of formalism | 12
- disclaimer | 9
- display command, in LP | 45
- documentation for LSL, on-line | 14
- dynamic logic | 9

**E**
- ease of expression, vs. ease of proving consistency | 35
- Ehrich | 14
- Ehrig | 14
- Eide | 2
- emacs, using it to run LP | 42
- emacs, using with LP | 15
- enabling condition (guard) | 9
- ensures clause, in a BISL | 57
- enumeration clause, position of | 17
- equational logic | 23
- equational logic, basis of LSL | 14
- equations, what to write | 31
- equivalence class | 27
- error character, in LP | 42
- error checking, specifying them | 60
- errors, specifying them | 60
- examples, in a trait | 20
- exceptions, to conversion claims | 22
- exempting and undefined terms | 22
- exempting clause | 22
- exempting clause, with conditions | 23
- existential quantifiers, in LSL terms | 38
- extensions | 32

**F**

| FAQ | 1 |
| FAQ maintainers | 1 |
| Feijs | 9 |
| Fetzer | 12 |
| files, produced by LSL checker with -lp switch | 44 |
| finiteness | 4 |
| finiteness, in a BISL specification | 62 |
| finiteness, specifying in a BISL | 60 |
| first-order logic | 34 |
| Fitzgerald | 2 |
| formal methods | 12 |
| formal methods, use of | 12 |
| formalism | 12 |
| formalism, disadvantages of | 12 |
| frame axiom, in a BISL | 57 |
| frame problem | 58 |
| framing procedure specifications, in a BISL | 57 |
| freely-generated sort | 27 |
| frequently asked questions | 1 |
| frequently asked questions, list maintainers | 1 |
| ftp address for LP | 42 |
| ftp address for LSL checker | 15 |
| ftp address, for LSL handbooks | 37 |
| ftp, address for FAQ | 1, 11 |
| fun | 13 |
| function, BISL specification and traits | 54 |
| funding credit | 2 |
funny symbols in LSL, input for .......................... 38
Futatsugi ........................................ 14, 16

G

game, thinking of LP as a ......................... 40
Garland ........................................ 2, 13, 35, 40
GCIL, where to get it .............................. 3, 9
GCIL, generated by clause, and generators ...... 32
generated by clause, using ......................... 19
generated freely by clause ........................ 28
generated, specifying ............................. 19
generator .......................................... 31
generator, free .................................... 27
Generic Concurrent Interface specification
Language .......................................... 3
GIF file, for Larch .................................. 11
glyphs, for special symbols in LSL ............... 38
Goguen ............................................ 14, 16, 17
Gries ............................................... 25
guarantee, specifying called procedure's in a BISL
................................................... 57
Guaspari .......................................... 27
guiding a proof, in LP .............................. 46
Guttag ............................................ 2, 3, 7, 10, 11, 12, 13, 14, 31, 40

H

Hall .............................................. 12
handbook, of Guttag and Horning ................. 7
handbooks, of LSL traits, where to find them ... 37
Hayes ............................................ 8
help card, for LP .................................. 42
Hinchey ........................................... 12
history command, in LP ............................ 45
Hoare ............................................ 3, 55, 56, 60
home page, for Larch .............................. 11
Horning ........................................... 2, 3, 5, 7, 10, 11, 13, 14, 31, 38
how did I get here?, in LP ......................... 45
HTML, conversion from LSL ....................... 39
hypertext, from LSL text .......................... 39

I

ill-defined, operator specification ..................... 27
immunity, when to set it on ........................ 47
immutable ....................................... 55
implies clause, position of ........................ 17
Larch, comparison to other specification languages ............................................. 6
Larch, general questions about ............................................................. 3
Larch, history ......................................................... 10
Larch, name .......................................................... 10
larch-interest, archives .......................................................... 11
larch-interest, mailing list ......................................................... 11
Larch/Ada, references for ............................................................. 3
Larch/Ada, where to get it ............................................................. 53
Larch/C++, references for ............................................................. 3
Larch/C++, where to get it ............................................................. 53
Larch/CLU, references for ............................................................. 3
Larch/CORBA ............................................................. 3
Larch/Generic ............................................................. 3
Larch/ML ............................................................. 3
Larch/Smalltalk, references for ............................................................. 3
Larch/Smalltalk, where to get it ......................................................... 53
LaTeX2e, macros for LSL symbols ....................................................... 38
LaTeX2e, with larch.sty ............................................................. 38
layer, synonym for tier ............................................................. 4
lazy evaluation, not needed in LSL ....................................................... 24, 25
LCL ............................................................. 7
LCL, references for ............................................................. 3
LCL, where to get it ............................................................. 53
LDE ............................................................. 15
learn, what to read to learn about Larch .................................................. 11
Leavens ............................................................. 23, 25
Lemma ............................................................. 47
lemmas ............................................................. 20
Lerner ............................................................. 9
Lescanne ............................................................. 2
LIL, aka a BISL ............................................................. 52
Lipton ............................................................. 12
Liskov ............................................................. 10, 11, 12
list of LP commands ............................................................. 42
literals, proving facts about in LP ....................................................... 48
literate programming for LSL ............................................................. 39
LM3, references for ............................................................. 3
Loeckx ............................................................. 14
logic, behind LSL ............................................................. 14
logic, classical in LSL ............................................................. 25
logic, first-order equational ....................................................... 23
logo, for Larch ............................................................. 11
loose semantics ............................................................. 23
LP ............................................................. 16
LP, as a game ............................................................. 40
LP, automatic proof techniques ....................................................... 45
LP, backspace character ............................................................. 42
LP, bearings, getting them ............................................................. 45
LP, bibliographic references for ....................................................... 41
LP, consistency proofs ............................................................. 48
LP, erase character ............................................................. 42
LP, finding your way ............................................................. 45
LP, ftp address for ............................................................. 42
LP, getting input from an LSL trait ..................................................... 44
LP, guiding a proof ............................................................. 46
LP, how to prove something ............................................................. 46
LP, how to use to check LSL traits ..................................................... 43
LP, need to use with LSL? ............................................................. 43
LP, philosophy of ............................................................. 40
LP, pitfalls ............................................................. 48
LP, quit character ............................................................. 43
LP, rubout character ............................................................. 42
LP, starting it ............................................................. 44
LP, steps in proof interaction ............................................................. 40
LP, tips for proofs ............................................................. 46
LP, use in checking traits ............................................................. 20
LP, use it ............................................................. 33
LP, useful for debugging ............................................................. 40
LP, useful for proof checking ............................................................. 40
LP, what it is ............................................................. 40
LP, what to do when you’re stuck ..................................................... 47
LP, when is it done? ............................................................. 45
LP, world-wide web address for ....................................................... 42
LSL ............................................................. 3
LSL checker produced files ............................................................. 44
LSL checker, ftp address for ............................................................. 15
LSL checker, latest beta ............................................................. 15
LSL checker, world-wide web address for ....................................................... 15
LSL symbols, input codes for ....................................................... 38
LSL, manual on-line ............................................................. 41
LSL, need to use LP? ............................................................. 43
LSL, object-oriented extensions to .................................................... 12
LSL, on-line information ............................................................. 14
LSL, purpose of traits ............................................................. 16
LSL, questions about ............................................................. 14
LSL, tips for writing equations ............................................................. 31
LSL, traits to reuse ............................................................. 37
LSL, translating into LP input ............................................................. 44
Concept Index

LSL, using LP to check traits .................................. 43
LSL, what it is .................................................. 14

M
macros, for LSL symbols with TeX or LaTeX ........... 38
Mahr ............................................................... 14
make active command, in LP ............................. 49
make, with LSL ................................................ 15
makefile, for LSL ................................................. 15
managing proofs in LP ...................................... 49
mantra, for BISL specifiers ............................... 61
Martin ............................................................ 2, 11
math, equals LSL specification .......................... 20
meaning, of a BISL specification ...................... 59
meaning, of an LSL trait .................................... 23
Meyer ............................................................... 2
mistake, in LP proof, how to correct .................... 48
mistakes, pitfalls in LSL .................................... 27
mixin, a kind of trait ......................................... 32
ML, Larch BISL for ............................................ 3
model, of a trait ................................................. 23
Modet ............................................................. 14
modified .......................................................... 57
modifies clause .................................................. 7
modifies clause, forgetting .................................. 62
modifies clause, in a BISL ................................. 57
modifies, vs. deallocation .................................. 58
Modula-3, Larch BISL for .................................. 3
Moore ............................................................. 35
Mosses ........................................................... 14
motto .............................................................. 20
mutable .......................................................... 55
mutation .......................................................... 4
myths, of formal methods .................................... 12

N
name, origin of Larch .......................................... 10
negative numbers, in LSL .................................... 33
newsgroup, for Larch ......................................... 11
noninteractive development of proofs in LP ........... 49
nostandard values, preventing ............................ 19
normalization .................................................... 45
nuclear ............................................................ 39
NSF ................................................................. 2
numbers, proving facts about in LP ..................... 48

O
OBJ ................................................................. 16
object-orientation ............................................. 12
object-oriented BISLs ....................................... 12
object-oriented, extensions to LSL ..................... 12
objects, vs. variable names in a modifies clause ........ 58
obligation, specifying caller’s in a BISL ............... 56
observer .......................................................... 31
OO BISLs ......................................................... 12
OO, extensions to Larch ...................................... 12
OO, extensions to LSL ........................................ 12
operator parameter, assumptions about .................. 17
operator, incompletely specified ......................... 25
operator, specifying uniquely-defined ................... 21
operators, are total functions ............................. 25
orientation, finding it in LP ................................ 45

P
parameter, assumptions about ........................... 17
partial correctness ............................................. 57
partial functions, modeling in LSL ...................... 25
partial functions, not in LSL .............................. 25
partial specification in LSL ............................... 25
partial specification, in LSL ............................... 25
partiality .......................................................... 4
partiality, specifying in a BISL ........................... 60
partitioned by clause, using ............................... 18
partitioned by, and a pitfall for LSL .................... 28
philosophy behind LP ......................................... 40
pitfall, bad recursion ........................................ 26
pitfalls, for BISL specifiers ............................... 61
pitfalls, for LP users ......................................... 48
pitfalls, for LSL ................................................ 9
pitfalls, for LSL specifiers ................................. 27, 29
post-state ........................................................ 56
postcondition, in a BISL ..................................... 57
pre-state .......................................................... 56
precondition, forgetting ..................................... 62
precondition, in a BISL ..................................... 56
precondition, in Z and LSL ............................... 9
precondition, omitted ....................................... 56
predicate .......................................................... 56
procedure specification, for partial procedures ..... 60
procedure specification, interpretation of ............. 57
procedure, BISL specification and traits .............. 54
proof checking, a use of LP ............................... 40
proof, by induction ............................................ 19
<table>
<thead>
<tr>
<th>Concept</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>proof, of implications in an included trait</td>
<td>51</td>
</tr>
<tr>
<td>proof, what to do in LP</td>
<td>46</td>
</tr>
<tr>
<td>proofs in LP, tips on how</td>
<td>46</td>
</tr>
<tr>
<td>proofs involving decimal literals</td>
<td>48</td>
</tr>
<tr>
<td>PVS</td>
<td>41</td>
</tr>
<tr>
<td>quantifiers, in LSL terms</td>
<td>38</td>
</tr>
<tr>
<td>quantifiers, vs. partitioned by</td>
<td>18</td>
</tr>
<tr>
<td>random, proof techniques in LP, no!</td>
<td>46</td>
</tr>
<tr>
<td>reading, what to read on Larch</td>
<td>11</td>
</tr>
<tr>
<td>recording proofs in LP</td>
<td>49</td>
</tr>
<tr>
<td>recursion, in LSL</td>
<td>26</td>
</tr>
<tr>
<td>recursion, pitfalls</td>
<td>27</td>
</tr>
<tr>
<td>redundancy, for debugging in a BISL</td>
<td>59</td>
</tr>
<tr>
<td>redundancy, in a trait</td>
<td>20</td>
</tr>
<tr>
<td>replay, of proofs in LP</td>
<td>49</td>
</tr>
<tr>
<td>requires, in a BISL</td>
<td>56</td>
</tr>
<tr>
<td>requires, clause, omitted</td>
<td>56</td>
</tr>
<tr>
<td>RESOLVE, and two tiers</td>
<td>4</td>
</tr>
<tr>
<td>respecting, equivalence classes</td>
<td>27</td>
</tr>
<tr>
<td>resume command in LP, picking up from a cancel</td>
<td>48</td>
</tr>
<tr>
<td>reuse of theorems from an implies section, in LP</td>
<td>51</td>
</tr>
<tr>
<td>reuse, and completeness in LSL</td>
<td>25</td>
</tr>
<tr>
<td>reuse, in LSL</td>
<td>37</td>
</tr>
<tr>
<td>reuse, of LSL traits</td>
<td>32</td>
</tr>
<tr>
<td>rubout character, in LP</td>
<td>42</td>
</tr>
<tr>
<td>Ruby</td>
<td>2</td>
</tr>
<tr>
<td>Rus</td>
<td>2</td>
</tr>
<tr>
<td>safety vs. speed in procedure specifications</td>
<td>56</td>
</tr>
<tr>
<td>schema calculus, in Z</td>
<td>8</td>
</tr>
<tr>
<td>schema, in Z</td>
<td>8</td>
</tr>
<tr>
<td>Schmidt, David</td>
<td>26</td>
</tr>
<tr>
<td>Schneider</td>
<td>25</td>
</tr>
<tr>
<td>searching for inconsistency with LP</td>
<td>48</td>
</tr>
<tr>
<td>sections, of a trait</td>
<td>17</td>
</tr>
<tr>
<td>semantics, loose</td>
<td>23</td>
</tr>
<tr>
<td>semantics, of a BISL specification</td>
<td>59</td>
</tr>
<tr>
<td>semantics, of procedure specifications in a BISL</td>
<td>57</td>
</tr>
<tr>
<td>shell principle</td>
<td>35</td>
</tr>
<tr>
<td>short-circuit evaluation, not needed in LSL</td>
<td>24, 25</td>
</tr>
<tr>
<td>side-effects</td>
<td>4</td>
</tr>
<tr>
<td>silver book</td>
<td>11</td>
</tr>
<tr>
<td>Smalltalk-80, Larch BISL for</td>
<td>3</td>
</tr>
<tr>
<td>SML, Larch BISL for</td>
<td>3</td>
</tr>
<tr>
<td>sort</td>
<td>16</td>
</tr>
<tr>
<td>sort parameter, assumptions about</td>
<td>17</td>
</tr>
<tr>
<td>sort, incompletely specified</td>
<td>25</td>
</tr>
<tr>
<td>sort, vs. type</td>
<td>16</td>
</tr>
<tr>
<td>specifand, of a BISL</td>
<td>59</td>
</tr>
<tr>
<td>specification, formal</td>
<td>12</td>
</tr>
<tr>
<td>specification, informal</td>
<td>12</td>
</tr>
<tr>
<td>spiderweb</td>
<td>39</td>
</tr>
<tr>
<td>Spivey</td>
<td>8</td>
</tr>
<tr>
<td>Sridhar</td>
<td>2</td>
</tr>
<tr>
<td>starting LP</td>
<td>44</td>
</tr>
<tr>
<td>starting over, in LP</td>
<td>48</td>
</tr>
<tr>
<td>state changes, how to specify in LSL</td>
<td>30</td>
</tr>
<tr>
<td>stateless, LSL is</td>
<td>29</td>
</tr>
<tr>
<td>steps for checking LSL traits with LP</td>
<td>44</td>
</tr>
<tr>
<td>steps in a proof with LP</td>
<td>40</td>
</tr>
<tr>
<td>stuck, what to do when you’re</td>
<td>47</td>
</tr>
<tr>
<td>summary, of LP commands</td>
<td>42</td>
</tr>
<tr>
<td>symbols, LSL input for</td>
<td>38</td>
</tr>
<tr>
<td>syntactic sugar, the modifies clause</td>
<td>58</td>
</tr>
<tr>
<td>syntax, for handling errors</td>
<td>61</td>
</tr>
<tr>
<td>Tan</td>
<td>13</td>
</tr>
<tr>
<td>TeX, macros for LSL symbols</td>
<td>38</td>
</tr>
<tr>
<td>Thatcher</td>
<td>14, 16</td>
</tr>
<tr>
<td>theorems</td>
<td>20</td>
</tr>
<tr>
<td>Theorems file</td>
<td>44</td>
</tr>
<tr>
<td>theory</td>
<td>16</td>
</tr>
<tr>
<td>theory police, nonexistence of</td>
<td>25</td>
</tr>
<tr>
<td>theory, of a trait</td>
<td>23</td>
</tr>
<tr>
<td>think, good advice for LP</td>
<td>46</td>
</tr>
<tr>
<td>think, in LP, yes!</td>
<td>46</td>
</tr>
<tr>
<td>tier</td>
<td>3, 4</td>
</tr>
<tr>
<td>tiers, advantages of</td>
<td>4</td>
</tr>
<tr>
<td>tiers, disadvantages of</td>
<td>4</td>
</tr>
<tr>
<td>timeless, LSL is</td>
<td>29</td>
</tr>
<tr>
<td>tips, for making traits easier to read and reuse</td>
<td>32</td>
</tr>
<tr>
<td>tips, for reusing traits</td>
<td>32</td>
</tr>
<tr>
<td>tips for LSL, see also motto and mantra</td>
<td>33</td>
</tr>
<tr>
<td>tips, for BISL specifiers</td>
<td>63</td>
</tr>
</tbody>
</table>
tips, for LSL specifiers .................................. 31
tips, for proving things with LP ..................... 46
total correctness ...................................... 57
total functions, in LSL ................................. 25
trait ...................................................... 8, 16
trait in a Larch/Smalltalk .............................. 54
TRAIT in Larch/Ada ..................................... 54
trait, purpose of ....................................... 16
trait, sections of ....................................... 17
trait, uses of .......................................... 16
trait, vs. ADT .......................................... 55
TRAITS in LM3 ........................................... 54
traits, and BISL specification .......................... 54
traits, don’t write from scratch ....................... 37
trashes clause .......................................... 58
trashes clause, forgetting ............................... 62
trashing, a variable .................................... 58
trashing, an object ..................................... 58
tree ........................................................ 10
tree, pictures of Larch .................................. 11
trick, of old mathematicians ............................ 47
tricks, for specifying state changes .................... 30
tuple clause, position of ................................ 17
tuple shorthand ......................................... 7
typing operator symbols in LSL ....................... 38

U
undefined terms ......................................... 22
undefined terms, in LSL ............................... 24
undefined, and exemptions ............................. 22
undefined, in logic ..................................... 9
undefinedness vs. underspecification ................. 23
undefinedness, in terms ................................ 25
underspecification ..................................... 22, 24
underspecification, and conversion ................... 22
underspecify, defined .................................. 25
uniform resource locator, for FAQ .................... 1, 11
uniform resource locator, for Larch ................... 11
uniform resource locator, for LP ...................... 42
uniform resource locator, for LSL ..................... 15
union clause, position of ................................ 17
union shorthand ........................................ 7
union, in LSL ............................................ 8
uniquely-defined operator, specifying in LSL ....... 21
universal quantifiers, within LSL terms .............. 38
URL, address for FAQ .................................. 1, 11
URL, address for Larch ................................. 11
URL, for LP ............................................ 42
URL, for LSL ........................................... 15
uses in LCL and Larch/C++ ............................ 54
using in Larch/ML ..................................... 54

V
value, abstract .......................................... 55
variable names vs. objects in a modifies clause .. 58
VDM, and two tiers ...................................... 4
VDM-SL .................................................. 4, 6
VDM-SL vs. Larch BISLs ............................... 7
VDM-SL vs. LSL ......................................... 6
verification, formal ..................................... 12
VHDL, Larch BISL for .................................. 3, 53
Vijayaraghavan ........................................... 2
vocabulary .............................................. 16
VSPEC, references for ................................. 3
VSPEC, where to get it .................................. 53

W
Wagner .................................................... 14, 16
weakly-specified function, defined ................... 25
well-defined, operator specification .................... 27
when clause ............................................. 9
where am I?, in LP ..................................... 45
Windows 95, executables for the LSL checker .. 15
Wing ...................................................... 2, 3, 4, 11, 12, 23, 25, 52
Widner .................................................... 7
workshop, on Larch ..................................... 11
world-wide web address, for LSL handbooks .. 37
world-wide web, address for FAQ .................... 1, 11
world-wide web, address for Larch ................. 11
world-wide web, address for LP ..................... 42
world-wide web, address for LSL .................... 15
world-wide web, from LSL ............................. 39
WWW source, from LSL texts ......................... 39

Z
Z .......................................................... 4, 8
Z vs. Larch BISLs ....................................... 9
Z vs. LSL ............................................... 8
Zilles ..................................................... 10
Table of Contents

Introduction .................................................. 1

Acknowledgements ........................................... 2

1 Larch and the Larch Family of Languages ........... 3
   1.1 What is Larch? What is the Larch family of specification
       languages? .................................................. 3
   1.2 Why does Larch have two tiers? ........................... 4
   1.3 What is the difference between LSL and a Larch BISL? .... 5
   1.4 How does Larch compare to other specification languages? .... 6
       1.4.1 How does Larch compare to VDM-SL? ............ 6
       1.4.2 How does Larch compare to Z? ..................... 8
       1.4.3 How does Larch compare to COLD-K? ............ 9
   1.5 What is the history of the Larch project? ............... 10
   1.6 What is the origin of the name Larch? ................... 10
   1.7 Where can I get more information on Larch and Larch
       languages? .................................................. 11
   1.8 Is there an object-oriented extension to Larch? .......... 12
   1.9 What is the use of formal specification and formal methods?
       ............................................................... 12

2 The Larch Shared Language (LSL) ................. 14
   2.1 What is the Larch Shared Language (LSL)? ............... 14
   2.2 Where can I find information on-line about LSL? ......... 14
   2.3 Where can I get a checker for LSL? ...................... 15
   2.4 Do you have a good makefile to use with the LSL checker?
       ............................................................... 15
   2.5 What is a sort? .............................................. 16
   2.6 What is the purpose of an LSL trait? What is a trait used
       for? .......................................................... 16
   2.7 What are the sections of an LSL trait? ................. 17
   2.8 What is the difference between assumes and includes? ... 17
   2.9 How and when should I use a partitioned by clause? ...... 18
   2.10 How and when should I use a generated by clause? ...... 19
   2.11 When should I use an implies section? .................. 20
   2.12 How and when should I use a converts clause? .......... 21
   2.13 How and when should I use an exempting clause? ....... 22
       2.13.1 Does exempting some term make it undefined? ... 22
       2.13.2 How can I exempt only terms that satisfy some
           condition? ................................................. 23
   2.14 What is the meaning of an LSL specification? .......... 23
   2.15 How does LSL handle undefined terms? ................ 24
2.16 Is there support for partial specifications in LSL? 25
  2.16.1 Can I specify a partial function in LSL? 25
  2.16.2 Do I have to specify everything completely in LSL? 25
2.17 Can I define operators using recursion? 26
2.18 What pitfalls are there for LSL specifiers? 27
2.19 Can you give me some tips for specifying things with LSL? 31
2.20 How do I know when my trait is abstract enough? 33
2.21 Is there a way to write a trait that will guarantee consistency? 34
2.22 Do I have to write all the traits I am going to use from scratch? 37
2.23 Where can I find handbooks of LSL traits? 37
2.24 How do I write logical quantifiers within an LSL term? 38
2.25 Where can I find LaTeX or TeX macros for LSL? 38
2.26 How do I write some of those funny symbols in the Handbook in my trait? 38
2.27 Is there a literate programming tool for use with LSL? 39
2.28 Is there a tool for converting LSL to hypertext for the web? 39

3 The Larch Prover (LP) 40
3.1 What is the Larch Prover (LP)? 40
3.2 What kind of examples have already been treated by LP? 41
3.3 How does LP compare with other theorem provers? 41
3.4 Where can I get an implementation of LP? 42
3.5 Is there a command reference or list of LP commands? 42
3.6 Can I change the erase character in LP? 42
3.7 How do I interrupt LP? 43
3.8 Do I need to use LSL if I use LP? 43
3.9 Do I need to use LP if I use LSL? 43
3.10 How do I use LP to check my LSL traits? 43
3.11 What is the use of each kind of file generated by the LSL checker? 44
3.12 If LP stops working on my input is it all correct? 45
3.13 How do I find out where I am in my proof? 45
3.14 What proof techniques does LP attempt automatically? 45
3.15 Can you give me some tips on proving things with LP? 46
3.16 What pitfalls are there for LP users? 48
3.17 How do I prove that my theory is consistent with LP? 48
3.18 How can I backtrack if I made a mistake in my proof? 48
3.19 How do I prove something like $0 \leq 2$ in LP? 48
3.20 How can I develop a proof that I can replay later? 49
3.21 Do I have to reprove everything in an included trait? 51
3.22 Does LP use assertions in the implies section of an included trait? 51
4 Behavioral Interface Specification Languages (BISLs) .......................... 52

4.1 What is a behavioral interface specification language (BISL)? .................................................. 52
4.2 Where can I get a BISL for my favorite programming language? .............................................. 52
4.3 Do I need to write an LSL trait to specify something in a BISL? ................................................. 53
4.4 What is an abstract value? ................................................. 54
4.5 What do mutable and immutable mean? ......................... 55
4.6 If I specify an ADT in a BISL do I prove it is the same as the trait? ............................................... 55
4.7 What does requires mean? ............................................. 56
4.8 What does ensures mean? ............................................. 57
4.9 What does modifies mean? ........................................... 57
4.10 What does trashes mean? ........................................... 58
4.11 What does claims mean? ............................................ 59
4.12 What is the meaning of a specification written in a BISL? ......................................................... 59
4.13 How do I specify something that is finite or partial? ...... 60
4.14 How do I specify errors or error-checking? .................. 60
4.15 How do I specify that all the values of a type satisfy some property? .......................................... 61
4.16 What pitfalls are there for BISL specifiers? ................. 61
4.17 Can you give me some tips for specifying things in a BISL? ...................................................... 63

Bibliography .......................................................... 65

Example Index ......................................................... 74

Concept Index ......................................................... 75