The Larch Shared Language: Some Open Problems

James J. Horning*

Digital Equipment Corporation, Systems Research Center,
130 Lytton Avenue, Palo Alto, CA 94301, USA

Abstract. The Larch Shared Language for the specification of abstract data types has evolved over a number of years from a simple algebraic language to one that is both more complicated and more useful. This talk reviews some of its major design decisions and then discusses some of the design issues and remaining open problems—most of which are consequences of the same decisions that contribute to LSL's good properties.

1 Introduction

Larch [3], [5] is a multi-site[2] project exploring methods, languages, and tools for the practical use of formal specifications. A distinctive feature of the Larch family of languages is that they support a *two-tiered* style of specification. Each specification has components written in two languages: one language that is designed for a specific programming language and another language that is independent of any programming language. The former kind are *Larch interface languages*, and the latter is the *Larch Shared Language* (LSL).

The Larch Shared Language is similar to many of the algebraic specification languages discussed at this workshop. It has been relatively stable for several years, and we are reasonably happy with it in practice. However, there are still a number of areas in which we see room for improvement. I am happy to share these opportunities with this group, since I am unlikely to tackle them any time soon. I would welcome your involvement; it may be that some of your work could usefully be applied, and it may be that some of the Larch work could be incorporated into your own research.

This will be a very Larch-centric talk. There have been continual interactions between members of the Larch Project and other members of the Abstract Data Type and Formal Methods communities, but I won't document and discuss them here.

2 Major Decisions in the Design of LSL

Before discussing LSL's open problems, I will review some key design decisions that made the language what it is today,[3] since they tend to constrain the space of possible

* Current address:
Silicon Graphics, 2011 N. Shoreline Blvd., MS 178, Mountain View, CA 94043, USA; e-mail: horning@sgi.com; URL: http://reality.sgi.com/horning/home.html; O– .

[2] Much of the foundational work was done in the Systematic Programming Development Group at MIT's Laboratory for Computer Science and at Digital's Systems Research Center.

[3] For a more extended discussion, see [4].

solutions. Good language features are relatively easy to design in isolation, but a combination of good features is not necessarily a good language, since language features interact in subtle and hard-to-anticipate ways. In the words of Sannella and Wirsing, “A specification language is a commitment to a compatible combination of choices.” *The devil is in the details.*

2.1 Meta-decision: Avoid Research in Mathematics When Possible

We repeatedly encountered opportunities to tackle new mathematical problems. We almost always chose to evade them instead, and apply the time saved to considering the practical problems of writing and using specifications. So LSL lacks many cutting-edge features found in other languages. But we have often been surprised by how well our “low tech” substitutes work in practice for dealing with problems for which “high tech” features are advocated.

2.2 Decision: Use Algebra to Specify ADTs

I still have my notes from my introduction to abstract data types and to algebraic specifications. On October 3, 1973, at a workshop organized by Barbara Liskov, Steve Zilles gave a talk in which he discussed separating the specification of what a data type *means* from describing choices about its *representation*. The example he used was Integer Sets:

```
Operators
  Insert: Sets x Ints --> Sets
  Remove: Sets x Ints --> Sets
  Has:    Sets x Ints --> Bools
  Null:               --> Sets
Relations
  Has(Insert(s, i), j) = if i = j then True  else Has(s, j)
  Has(Remove(s, i), j) = if i = j then False else Has(s, j)
  Has(Null, j) = False
```

He argued that anything that could reasonably be called a set would have these operations and satisfy these equations, and conversely, that anything that had these operations and satisfied these equations should be accepted as an implementation of a set, regardless of the representation it used. He thoroughly convinced me—although there are actually some bugs (which I will discuss later) in this specification.

When I returned to the University of Toronto and shared this idea with my students, John Guttag took an immediate interest. We quickly saw that stacks were even easier to specify than sets, and queues were a bit harder—but not much harder. And most of the other data types and data structures that we looked at seemed to fall in that same range of difficulty. Furthermore, John noticed that the left hand sides of the equations we were writing all seemed to fall into a standard form: one operator applied to another. In fact, if we partitioned the operators of an ADT into *generators*, *extensions*, and *observers*, it seemed that we got an adequate set of equations by writing an axiom for each extension or observer applied to each generator, equating that term to a “simpler” term.

In the case of Integer Sets, `Null` and `Insert` are generators, `Remove` is an extension, and `Has` is an observer.[4]

In his thesis[2], John proved that any ADT with computable operations can be specified by axioms in this standard form (if "hidden" operators are allowed), and explored the issue of being "simpler" in some depth.

2.3 Decision: Loose Semantics

Another early decision was to use what is now called a *loose semantics* for the equations in a specification. That is, the *absence* of an equation from a specification provides no information about whether terms are distinct or equivalent. We realized that this meant that specifications could have non-isomorphic models, but we believed (and still believe) that it is an advantage to allow them.

Restricting specifications to those with isomorphic models:

- forces you to say too much,
- over-restricts implementations,
- makes specifications harder to extend, and
- is too error-prone.

For example, the Integer Set specification given above needs two more equations to make the initial model be equivalent to ordinary sets (which is what Steve intended):

```
Insert(Insert(s, i), i) = Insert(s, i)
Insert(Insert(s, i), j) = Insert(Insert(s, j), i)
```

Without these axioms, restriction to the initial algebra rules out many useful implementations of integer sets and defines a very unusual data type.[5] [6]

So we decided not to give initial or final algebras any special status. Instead, we included explicit language constructs to make it possible to write specifications whose models are all isomorphic when that is the specifier's intent (**generated by** and **partitioned by**).

2.4 Decision: Two-tiered Specification Strategy

We wanted to specify real programs in a variety of real programming languages. What should it mean to say that a program "correctly implements" a specification? Much work on algebraic specification has ignored this question.

What did it mean for a program to satisfy the Integer Set specification? If we had been interested only in a simple functional language, we might have required that each

[4] The equations above are not in the standard form, but it is an easy exercise for the reader to write the corresponding standard form equations.

[5] Work it out.

[6] One of the referees pointed out that I had forgotten two more necessary equations:

```
remove(null, j) = null
remove(insert(s, i), j) =
  if i = j then remove(s, i) else insert(remove(s, i), j) .
```

equation hold when the program's functions were substituted for the specification's operators. But this approach didn't seem to carry over to languages in which procedures changed program variables. We needed *another specification* to describe how the procedures changed program variables, in terms of the abstract values defined in the Integer Set specification. Fortunately, the classical precondition/postcondition style of procedure specification that had been developed for program verification seemed to work very well. Predicates used in these specifications could incorporate operators from ADT specifications.

This general approach seemed fairly satisfactory. But to give a precise definition of the relation *program* P *satisfies specification* S, we needed to take account of more than just program variables. We had to deal with parameter passing modes, storage allocation, aliasing, the type system, exception handling, concurrency, etc. This morass of "details"—which simply could not be ignored in practical specifications—quickly overshadowed the modest complexity of our ADT specification language. And the challenge of making it adequate to handle several real languages was truly daunting.

The answer was neither to give up on the approach nor to give in on complexity, but to separate concerns. We separated the *ADT tier*, which could be purely mathematical and abstract, from the *interface tier*, which would deal with specifying program interfaces (procedures, types, modules, etc.).

The ADT tier was totally independent of programming languages, so one language was enough for this tier[7]—its ADT specifications could be used with any programming language—and we called it the *Larch Shared Language* (LSL). We wanted to put most of the subtlety in the simplest language that would do the job, so this tier is where we planned to put all the *mathematically interesting* stuff. *Reusable specification components* are generally language-independent abstractions, so we expected to write most of them in LSL.

Interface languages were designed to deal with state, types, parameter passing, flow of control, errors, exceptions, and so on. Because programming languages differed in their choices for these, we found it helpful to further separate concerns by using a different interface language for each programming language. This allowed both the syntax and the semantics of each interface language to be closely matched to its programming language. For example, the precise notion of *type* came directly from the corresponding programming language. The collection of built-in types and type constructors were defined by a special LSL trait particular to that programming language. The notion of *exception* (or *signal*) was also defined by the programming language. The possibility of *threads* for concurrency could depend on the language or on the run-time library. The interface tier is where we put all the *messy, boring* stuff that happened to be essential to the specification of real programs.

From the perspective of this talk, the important point is that LSL did not have to deal with any complexities mandated by programming languages.

---

[7] But there is no fundamental reason why multiple languages could not be used on this tier, too.

2.5 Decision: Simple "Putting Together" Operations

Reading specifications is important. And people read syntactic objects (texts), rather than semantic objects (theories). So we generally thought of LSL's combining operators as operators on specifications, producing new specifications, rather than as operators on theories or models, producing new theories or models. To put it another way, we were more interested in structuring specifications than in structuring theories.

LSL's unit of specification is the *trait*. We defined inclusion and parameterization as syntactic operations on (the text of) traits. To us, this seemed easier to explain than operations on theories. LSL's combining operations apply to simple, tangible objects; all derived entities can be explicitly mechanically constructed and exhibited. Of course, there are corresponding induced operations on the associated theories, but focusing on the presentations made it easier to avoid such complications as parameterized theories and theory parameters.

Even so, the hardest issue was trait parameterization. We tinkered with it many times. An early version of LSL had only explicit lambda abstraction. We soon discovered that it was hard to get a trait's formal parameter list "right." If we kept it short, we often wished to substitute for a sort or operator that did not appear in the list; if we made it longer, we frequently didn't need to change most of the potential parameters, and supplied the formal names again for the actuals. This led us to abolish explicit parameter lists in a later version; all renaming was of the form "name1 **for** name2." But the restriction to explicit renaming also proved cumbersome. So we compromised with a design that allows the specifier to choose to rename either positionally or explicitly.

Although we originally allowed *hiding* as one of our operators on traits, we eventually removed it from LSL. "Hidden" operators cannot be completely hidden, since they must be read to understand the specification, and they are likely to appear in reasoning based on the specification.

We decided to limit the combining operations on specifications to the LSL tier; we would use combinations of LSL specifications in interface specifications, but we would not use interface specifications within LSL specifications.

2.6 Decision: Designed-in Redundancy for Checking Tools

One of the things we have learned by bitter experience is that even carefully-written specifications are at least as error-prone as carefully-written programs,[8] and that specifications that have not been rigorously and mechanically checked are no more likely to be right than programs that have not been compiled. My rule of thumb is that a good mechanical checking tool will find about one error per page of carefully hand-checked specifications.

One of our goals in designing LSL was to include redundancy that could be used to find more errors sooner. We felt that it would be important to check specifications incrementally as they were developed, rather than all at the end, for several reasons: we wanted to reuse and extend existing specifications; we wanted to catch specification errors as early as possible during development; and we wanted to check specifications that

[8] This is one reason why it is probably not desirable to mechanically generate programs from specifications, even if that ever becomes technically feasible.

were intentionally incomplete.[9] So we designed LSL with checking tools in mind, and redesigned it as the development of tools improved our understanding of the issues.

Some of the features of LSL that contribute to checkability are

- mandatory declaration,
- syntax and sort-checking,
- the requirement of consistency,[10]
- explicit assumptions,
- optional claims
  - about the (degree of) completeness of the axioms,
  - about implications of the axioms,

Much of this checking requires theorem-proving, which led us to develop a proof assistant (LP) specifically designed for this type of checking.

In early versions of LSL, we included two other forms of redundancy (**imports** and **constrains** clauses) to be used to assert that a theory was a *conservative extension* of another. We found that these constructs were difficult to explain and to use effectively. Furthermore, we did not know how to check them. (These reasons are probably all related.) So we dropped them from the language.

2.7 Decisions: Types and Sorts

In each interface language, "type" must be consistent with what it means in its programming language. However, in LSL, we needed a more basic notion of "sort" that could be used by all interface languages. More precisely, each type would be *based on* a sort, meaning that, in any model, all values of the type would be members of the carrier of the sort.[11]

We first tried the simplest non-trivial sort system we could think of: Sorts were distinct precisely when they had distinct names. We did not impose any structure on this global name space. There was no notion of sub- or super-sort. There was no notion of structured sorts, but we often used the sort names as stylized comments (e.g., `IntSet` or `CharSet`).

This simple scheme actually carried us quite a way, and let us focus on more pressing problems early on. It served us well in the development of an *LSL Handbook* [6]. Where it started to feel clumsy was in writing interface specifications that involved type constructors. `IntSet` and `IntArray` aren't too bad in an interface that involves only the corresponding types. But in bigger interfaces, with more types involved, the strategy of relying on mere conventions about the choice of names for sorts was less and less satisfactory.

[9] And even specifications that are intended to be complete seldom are!

[10] If a specification implies that `true = false`, *we* would like to be warned; even though this is difficult to check in general, many such contradictions can be found in practice. Designers of some other languages have decided that it is perfectly all right for a specification to imply that there is only one value of sort Bool.

[11] The correspondence is not one-one: Many types can be based on the same sort.

In our second try, we designed a simple structured name space, allowing *sort terms*, such as `Set(Int)`, `Set(Char)`, `Array(Int)`, `Pair(Stack(Int), Bool)`. However, we retained the simplicity of sort equivalence: Sorts were distinct precisely when they were distinct terms, and sort names were still global. I will return later to some of the problems this did not solve.

Sorts are used for three fairly distinct purposes in LSL:

- To resolve operator overloading.
- To enable detection of many errors in terms.
- To represent the carriers of the algebra in quantifiers and when specifying complete sets of generators or observers.

Thinking about the first and second led us to make operator and variable declaration (with sorts) mandatory. If we had been thinking more about the third, we would probably have made sort declaration mandatory, too. Sorts play an important role in proofs, especially proofs by induction. We cannot induct over arbitrary values, but only over the values of a particular sort.

2.8 Decision: True Functions

Because of its algebraic heritage, we assumed from the outset that any model of an LSL specification would have to interpret each operator as a single-valued, total function. We reject(ed) "non-deterministic functions" on the grounds that algebraic equations do not make sense in a system where equals cannot be freely substituted for equals.[12]

Our decision to insist on total functions was probably more controversial. We allowed operators to be underdetermined, and models to be non-isomorphic. But syntactically correct terms were never "undefined" or "erroneous" in the logic.[13]

Having read and heard lengthy discussions of partial algebras and partial logics, I have no regrets about evading this particular tar-pit. The requirement of totality can be a subtle source of surprises, but they do not seem to be more numerous, more subtle, or more dangerous, than the ones that come with partiality. And total functions are certainly the easiest to explain to users.

There is a certain amount of mess that can be pushed around, but not eliminated. In the words of Jim Thatcher (1977) "Mathematicians have not found an elegant way to deal with division by zero in 400 years, and there is no reason to expect that computer scientists will do so in 40."

2.9 Decision: First-Order Logic with Equality

The standard form of algebraic specifications discussed in section 2.2—in which each equation has a right hand side "less" than the left[14]—lends itself naturally to interpretation as term rewriting systems. There is a well-developed theory of term rewriting systems, building on the Knuth-Bendix *completion* algorithm. We originally had high hopes

[12] It is still possible to specify relations by functionally defining their characteristic predicates, thereby avoiding an abuse of functional notation.

[13] Of course, a specification may include sorts with explicit values to represent "undefined" or "error," but these values have no special properties in the logic.

[14] In some ordering on terms.

of using this for proving things about our specifications. However, although term rewriting is useful, we quickly learned that it is not sufficient because 1) K-B doesn't terminate for many cases of practical interest, and 2) in practice completed systems for rich ADTs (e.g., arithmetic or propositional logic) are neither efficient decision procedures nor useful sources of diagnostic information when a proof attempt fails.[15]

Similarly, our hopes for "inductionless induction" were dashed when we discovered that establishing the preconditions for its soundness is usually as hard as doing ordinary induction. Furthermore, we discovered that first-order quantification occurs naturally in many specifications. Skolemization can be used to transform quantified formulas into equational form,[16] but this process is more likely to obfuscate than to clarify. So we kept generalizing LP. The current version (3.1) supports full first-order logic with equality.

First-order logic fits well with our use of ADT specifications in precondition/postcondition specifications of procedures (in the interface language tier), and is, of course, the most familiar and widely used logical system.

2.10 Decision: Shorthands

We realized early on that we were repeatedly writing some very stylized LSL traits—almost cliches—for three common kinds of data structures: enumerations, labelled tuples (records), and discriminated unions (variants). We initially argued there was little harm in this, since the specifications were so easy to write, and to recognize and skip while reading. But, over time, we recognized this "boilerplate" as redundancy of the worst sort:

- The number of axioms could be non-linear in the size of the data structure.
- It required the invention of too many names.
- It invited careless writing that could introduce errors.
- Errors were unlikely to be noticed and corrected, since the readers would be skipping over the boilerplate.
- The mass of uninteresting detail obscured the interesting parts of specifications.
- It discouraged first-time readers and writers.

So we added special shorthands for these three idioms.[17] I will discuss later why the shorthands are still sources of difficulty.

2.11 Decision: Configurable Lexical Conventions

We needed to use LSL traits in interface specifications for programming languages that have very different lexical conventions (e.g., what constitutes a string? a comment? what are the operators? the identifier characters? the reserved words, if any?). So we could not "hardwire" any of these choices into LSL itself.

[15] Proof attempts practically all fail since a) most things we try to prove aren't true, and have to be debugged, and b) most proof sketches, even of valid theorems, must also be debugged.

[16] Although this preserves satisfiability, it does not, in general, preserve models.

[17] We have not reconsidered this decision since adding sort terms.

We designed a tiny *lexical customization language*, and required the LSL tools to read a customization (`.init`) file before processing traits. In many respects, this worked well, allowing us to write LSL traits for use with wildly incompatible programming languages.

2.12 Decisions: Syntax

Designing the syntax of a language always seems to be more difficult and controversial than designers expect.[18] It seems that the semantics represents the most important part of the language, but the syntax is what everyone sees, and everyone has opinions. LSL was no exception.

Most of the syntax was fairly straightforward. We reserved our keywords, punctuated liberally, and developed a grammar that was easily parsed—both by humans and by computers. The exception was the subgrammar for expressions (terms).

We had some ambitious goals for LSL's term syntax:

- Allow virtually all programming language expressions.
- Allow a wide variety of (linear) mathematical expressions:
  - infix: $x + y + z, \quad e \in S$.
  - bracketed: $\{x, y, z\}, \quad x[y]$.
  - mixfix: **if** $x < y$ **then** x **else** $y, \quad x \;\triangleleft\, x < y \triangleright\; y$.
- Allow terms to be parsed without seeing declarations, which may be in other traits.[19]

Experienced language designers will quickly realize that *these goals are incompatible!* So the questions were: which should we sacrifice first? and how small could we make the sacrifices?

We devised a rather elaborate syntax for terms, in which disambiguation depended on "correct" token classification, relying on the token customization mechanism discussed in the previous section. I will have more to say about syntax in section 3.5.

3 Issues and Open Problems

I have already hinted at some of the problems that followed from our design choices for LSL. Now I will discuss some of them in more detail, sketching possible solutions that I have thought about. However, be warned: I know the problems are all practical problems, but I do not know whether any of these solutions are practical. Compromises will undoubtedly be necessary to fit any solution smoothly into LSL, and the cost of these compromises should be carefully weighed.

3.1 Problem: Shorthands vs. Renaming

Although we expected LSL's enumeration, tuple, and discriminated union shorthands to eliminate much of the "boilerplate" in specifications, in practice we have not found

[18] Even designers who already know this.

[19] Or the terms may be in interface specifications, with the declarations in traits.

them as useful as we expected. Although the boilerplate was very stylized, it did include some variation, such as whether or not enumerations were ordered. Every time we made a choice when designing a shorthand, we ruled out some potential uses of the shorthand.

A more serious problem is that LSL's shorthand mechanism clashes with its renaming mechanism. Automatically generated operator names for enumerations, tuples, and discriminated unions must be explicitly and individually renamed when the corresponding components are.

For example,

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

which seems rather tidy, expands to

```
introduces
   [__, __]: E, S -> C
   __.hd: C -> E
   __.tl: C -> E
   set_hd: C, E -> C
   set_tl: C, S -> C
asserts
   C generated by [__, __]
   C partitioned by .hd, .tl
   ∀ e, e1: E, s, s1: S
     ([e, s]).hd == e;
     ([e, s]).tl == s;
     set_hd([e, s], e1) == [e1, s];
     set_tl([e, s], s1) == [e, s1];
```

a great savings. However, to rename the fields to `first` and `rest`, we would have to apply a renaming like

```
(__.first for __.hd, __.rest for __.tl,
 set_first for set_hd, set_rest for set_tl)
```

which greatly reduces the appeal of the shorthand.[20]

This problem may be a result of the decision to use one simple mechanism (renaming) for three purposes that many other languages deal with separately: *parameterization*, *fitting*, and *hiding*.

Alternatively, perhaps the "need" for shorthands is a symptom of a lack of expressive power in the base language. Maybe we should have asked ourselves instead: What would we have to add to the language so that we could just write traits for these three shorthands?

3.2 Issue: Parameterizing Operator Names

When a sort is renamed, it might be desirable to implicitly rename some of its operators, in addition to adjusting their signatures.

[20] One of the referees asks "Why not use a renaming of the form:

```
(tuple of (first, rest) for tuple of (hd, tl))
```

if this is what you want?" to which I can only respond that I have thought of renaming as "the substitution of one name for another" for so long that this form had never occurred to me.

"In Larch/C++ we treat = in assertions as a call to a trait function named `equal_as_T`. For each type there is a point-of-view of equality for that type. For example, consider pairs of integers, `IntPair`, and triples of integers, `IntTriple`. If we want `IntTriple` to be a subtype of `IntPair`, then we will need to define `equal_as_IntPair` for combinations of `IntTriple` and `IntPair` arguments. We would also define `equal_as_IntTriple` for `IntTriple` arguments. Now suppose we've defined a trait like this, but in C++ the name of the type is `PairInt` instead of `IntPair`. I want to rename `IntPair` to be `PairInt` by saying `IntPairTrait(PairInt` **for** `IntPair)`, but that won't change the name `equal_as_IntPair` to be `equal_as_PairInt`."—Gary Leavens

It may be that introducing structure into operator names, analogous to the structure for sort names, would solve this problem. But it would obviously require some syntactic invention; we cannot further overload parentheses to indicate structure in operator names.

This may be closely related to the solution of the shorthand problem.

3.3 Issue: Conservative Extension

Originally, LSL had a construct (**constrains**) to indicate which operators were being "defined" (or whose constraints were being augmented) by a group of axioms. But we did not know exactly what this "should" mean, or how to check it.

LSL also distinguished between a trait *including* another trait and *importing* it. The intuitive notion was that we did not want to accidentally change the theory of, say, Ints or Bools, by axioms in a trait that imported them. We dropped this distinction when we discovered that we did not have a precise meaning to go with this intuition. But the original intuition seems to have been valid. We do need some checkable redundancy here.

"One of the most problematic user traps in LSL is the ease with which specifiers can write inconsistent specifications. As we've learned, it's extremely hard to check consistency [in general]. Hence it's important to provide specifiers with a means for making claims (such as conservative extension) about consistency that introduce checkable proof obligations. Some claims about conservative extension are easy to check syntactically (e.g., extensions by explicit or primitive recursive definitions); others may require user-supplied proofs (e.g., via theory interpretation). We took checking conservative extension out of LSL because we thought it was too hard; we need to put something like it back in because it is too important to ignore." —Steve Garland

Several languages have a "definitional" or "shell" construct that allows operator definitions to be introduced in a way that guarantees that they do not introduce contradictions or allow the deduction of new properties of existing operators. Can this fit neatly into the LSL framework? I do not see any obvious problems, but it may be that certain restrictions would have to be imposed, at least in contexts where these constructs were used.[21]

[21] One of the referees points out that if the language is restricted to the standard form of axioms discussed in section 2.2, in many cases the semantic consistency of traits can be verified by syntactic checks. And axioms can always be omitted without loss of consistency. "Thus in many (practical) cases trait consistency can easily be established…If the standard form is complete

3.4 Issue: Arithmetic

There is no logical need to treat arithmetic types and operators specially. We have written handbook traits for them that are semantically adequate. But, particularly when it comes to proving things, it would be good to exploit many of the special properties of numbers that mathematicians have discovered over the centuries, just as we treat logical formulas specially—primarily for efficiency. Steve Garland asks, "Can we agree on hardwiring one or more theories of arithmetic? Can we do it in a way that does not force us to distinguish among 0:N, 0:Int, 0:Q, and 0:R?"

3.5 Issue: Operator Precedence

Probably the least satisfying aspect of LSL is its syntax for operators in expressions. We worked very hard to accommodate many of the most common mathematical and programming language notations. But the result is both too complex and too restrictive. It is one of the hardest things to teach about the current version of the language. ("Why do I need parentheses *there* ?")

This is an issue that did not surface while we were focusing on LSL, *per se*, and the LSL handbook; terms in axioms do not tend to be complex enough for the restrictions and the extra parentheses to be annoying. But it is a continual irritation in writing interface specifications, where expressions tend to be more complex, and where the programming language has already established expectations about the precedence of operators. Interface specifiers should be thinking about the programming language interface, not about the peculiarities of LSL.

As Steve Garland notes, "Users who want to reason about arithmetic will not be happy having to type expressions like

```
((3*(x^2)) - (4*(y^3))) < (10*(z/3))
```

instead of

```
3*x^2 - 4*y^3 < 10*z/3
```

- Should we fit some more symbols into the built-in precedence hierarchy? E.g, `+`, `*`, `-`, ...
- Should we give users control over precedence? If we do, the control needs to reside in the traits, not in an `.init` file. The meaning of a trait should not change if it is parsed with a different `.init` file."

Luca Cardelli has designed an elegant method for syntax extension that is rather general, dynamic (extensions can be introduced locally), well-scoped (including avoidance of bound variable capture), and not too hard to implement [1]. The principal restriction is that the (extended) grammar must be LL(1); more precisely, the parse yielded will be the first one found by recursive descent.

Perhaps we should define a much simpler LSL base grammar, and use Cardelli's extension mechanism to allow each trait to introduce syntax for the operators it introduces

and each type introduced has a **generated by** clause and a proved **partitioned by** clause, then semantic completeness follows; and any consistent extension is also a conservative extension. Why not make use of [these properties] in the Larch framework?"

and axiomatizes. This should make it easy to write each trait in its "most natural" syntax. However, I'm not sure how well such extensions will compose when traits are combined—especially, the resulting operator precedence. And is it wise to allow information in traits to control how expressions in interface specifications are parsed?

Maybe it would be better to let each interface language define its own translation to LSL terms from expressions in its own extension of its programming language's syntax.

3.6 Problem: Hidden `.init` Files

Although `.init` files solved a knotty problem for us, they also turned out to be a subtle source of problems that were hard to diagnose. An `.init` file can, by reclassifying tokens, completely change the interpretation of a trait, or turn a valid trait into one that will not even parse. This is dangerous and can be very confusing, since, like most configuration files, .init files tend to be ignored by both writers and readers.

This information is logically a part of each trait, but most of the time we don't want to look at it. Where should we put it?

3.7 Issue: Subsorting

LSL's treatment of sorts as disjoint can be quite annoying when dealing with ADTs that are "natural" subtypes of other types (e.g., Prime $\subset$ Positive $\subset$ Natural $\subset$ Integer $\subset$ Rational). There has been a lot of promising work on algebraic specifications with subsorting, and it would be very attractive to introduce subsorts into LSL. The issue is how to do it without complicating sort-checking too much.

I have made a sketchy proposal to deal with subsorting syntactically, based on a simplification of the Modula-3 type system. The key idea is to introduce a partial order on sorts (`<:`), and a new kind of assertion,

subsort `T <: U`

This would give three deduction schemata for

- operator signature subsorting on domain sorts: for every `U`-operator, there is implicitly a corresponding `T`-operator,
- quantifier subsorting: any `U`-variable can range over `T`, and
- widening: every `T`-term is also a `U`-term.

It would, however, not give schemata for covariance on range sorts or for implicit narrowing.

Operators, including constants, should be declared with their "smallest" range and "largest" domain sorts. The schemata ensure that it is always possible to use a term with a specific sort in a context that expects one of its supersorts. An operator application would be given the signature with the sorts of the domains exactly matching the (syntactic) sorts of its arguments—this would associate it with the strongest applicable theory for an operator with that symbol.[22]

The details have not been worked out. For example, perhaps we would need a rule that if an operator has two applicable overloadings that differ only in that the range of

[22] It is OK to require an exact match here, because of implicit operator signature subsorting.

one is a subsort of the range of the other, then the subsort is chosen.[23] The implications for the rest of the language and for LP would need to be investigated.

Semantic (predicative) sorts would certainly be more powerful, allowing us to define subsorts like Prime by characteristic predicates. But they turn sort checking into theorem proving,[24] and would undoubtedly make a real mess of term matching and rewriting.

3.8 Issue: Total Functions

The decision to stick to total functions was made very early in the design of LSL. It allowed us to evade a knotty set of problems, and concentrate our efforts on things that seemed more urgent. I am still not dissatisfied with it, but a lot of smart people have suggested we should revisit this issue.

In a typical LSL specification of the rationals, `1/0` isn't undefined; the specification just doesn't give any axioms that let you prove that `1/0` is equal to some fraction with a non-zero denominator.[25] We have relegated the issue of partiality (and what to do about it) to the interface language tier. This choice is quite conventional in algebra and logic, somewhat less so in computer science, where computability is an issue. Can we get away with it?

At one level, the answer is surely Yes, because we know how to replace any partial function with a total relation.[26] But there are pitfalls for the unwary specifier here:

– It is easy to accidentally introduce an inconsistency. (Remember all the high school algebra "paradoxes" that depend on a hidden division by zero?)
– The specifier must be careful not to give a **generated by** clause that doesn't generate these non-standard terms.
– The verifier must take the "extra" generators into account in proofs by generator induction.

Of course, if we added subsorting, that would allow us to introduce functions that are total over subsorts.

3.9 Issue: Composability

"The mechanisms for composing specifications in LSL are too primitive. They fail to address issues such as information hiding or interfaces, which we know play important roles in composing programs. As a result, it is too easy to produce inconsistent specifications by composing consistent ones.

"For example, two specifications may each define auxiliary operators (with common signatures, but different properties) to simplify some complicated definitions. To avoid

[23] One referee suggests that we could just require regularity, as in Order-Sorted Algebras.
[24] At least, if we insist on finding the semantically optimal sort.
[25] The *LSL Handbook*'s Integer trait does not specify a division operator for integers. This operator will have different semantics in different programming languages—and not just in the treatment of division by zero. Also, it would be awkward to deal with `1/0` as a separate generator in every induction over the integers. However, in the rationals we cannot escape `1/0`.
[26] At a hideous notational cost!

introducing an inconsistency when composing these specifications, the specifier must detect and resolve any name clashes between these operators. It would be better to hide the existence of such operators by appropriate interface mechanisms."—Steve Garland

3.10 Issue: Underlying Logic

"Many other specification and proof environments offer higher-order logics, order-sorted logics, logics of partial functions, … It is easier to write many specifications in these logics, but they are not as simple or as familiar as first-order logic. How do we strike an appropriate balance that preserves simplicity, facilitates expression, lessens the likelihood of error, and permits effective proof support?"—Steve Garland

4 Concluding Remarks

Language design is hard, and it's certainly not finished until *somebody else* implements your language, too.

The hardest problems in language design come from interactions of features, so

- keep everything as simple as possible as long as you can,
- know what the tradeoffs are, and
- don't assume you can "just" add another feature.

One reason I still have hope for LSL is that it has not yet fallen into the clutches of a standardization committee. :-)

Our most urgent problems do not concern refining our specification languages; they concern getting people to use them.

Acknowledgments[27]

Without the continuous contributions of John Guttag, there would never have been a Larch Shared Language. I am also indebted to the work of many other participants in the (loosely defined) Larch Project, including M. Abadi, G. Feldman, S.J. Garland, K.D. Jones, G. Leavens, P. Lescanne, W.M. McKeeman, A. Modet, J.B. Saxe, J. Wild, J.M. Wing, and J. Zachary. Ideas, criticism, and inspiration have also come from many researchers outside the project, notably J.R. Abrial, O.-J. Dahl, C.A.R. Hoare, B. Liskov, C.G. Nelson, and S. Zilles. IFIP Working Group 2.3 (Programming Methodology) provided an excellent sounding board over the years. Crucial support for the research came from the University of Toronto, the National Research Council of Canada, the Xerox Palo Alto Research Center, Digital Equipment Corporation, the National Science Foundation, the Defense Advanced Research Projects Agency, and the Massachusetts Institute of Technology.

[27] Because of the thoughtful comments of three anonymous referees, this paper has many fewer errors of detail, and several more footnotes.

References

1. Cardelli, Luca: *An Implementation of* $f_{<:}$. Digital Equipment Corporation, Systems Research Center, Palo Alto, Report 97, 1993.
2. Guttag, John V.: *The Specification and Application to Programming of Abstract Data Types.* Ph.D. Thesis, Department of Computer Science, University of Toronto, 1975.
3. Guttag, John V., Horning, James J., with Garland, S.J., Jones, K.D., Modet, A., Wing, J.M.: *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science, ISBN 0-387-94006-5, ISBN 3-540-94006-5, 1993.
4. Guttag, John V., Horning, James J., Modet, Andrés: *Report on the Larch Shared Language: Version 2.3.* Digital Equipment Corporation, Systems Research Center, Palo Alto, Report 58, 1990.
5. *Larch Home Page*.
   URL: `http://reality.sgi.com/horning/larch-home.html`.
6. *An LSL Handbook*. Appendix A in [3];
   URL: `http://reality.sgi.com/horning/toc.html`.

This article was processed using the LaTeX macro package with LLNCS style