The Larch Shared Language: Some Open Problems

Jim Horning
Digital Equipment Corporation
Systems Research Center
Palo Alto, California, USA

Slides for a talk given at the
11th Workshop on Abstract Data Types
Oslo
22 September 1995

LSL Open Problems

- Major Decisions in LSL
  - Algebra
  - “Putting together”
  - Two tiers
  - Redundancy and checking
  - Functions, types, and sorts
  - Syntax
- Open Problems and Issues in LSL
  - Renaming
  - Tokens and syntax
  - Arithmetic
  - Conservative extension
  - Subsorts
  - Composability
  - Logic

Decision:
Use Algebra to Specify ADTs

Steve Zilles, 1973

- Multi-sorted algebra

John Guttag, 1975

- Normal form for specification
  - partition operators
    * generators
    * extensions
    * observers
  - write an axiom for
    * each extension or observer applied to
    * each generator
    * equating it to a “simpler” term

Steve Zilles, October 3, 1973

A “type” is characterized by the behavior of its operations.

Example: 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
```

Decision: Loose semantics

Restriction to isomorphic models

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

So we did not give initial or final algebras any special status

- but included language constructs to make categorical specifications possible

Which Set Axioms Did Steve Zilles Forget?

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

Insert(Insert(s, i), i) = Insert(s, i)

Without these axioms, the initial algebra approach rules out many useful implementations of integer sets (and defines a very unusual data type).

Decision: Simple "Putting Together" Operations

Structured traits, rather than structured theories

- Trait inclusion and parameterization defined as operations on (the text of) traits

Seemed easier to explain

- Operations on simple, tangible objects
- All derived entities can be explicitly, mechanically constructed and exhibited

Of course, there are induced operations on the theories

- but focusing on the presentations made it easier to avoid such complications as parameterized theories

Any name in a trait can serve as a formal parameter
(be renamed)

(Eventually) no hiding operators

Decision: Semantics in First-Order Logic with Equality

Term rewriting is useful, but not sufficient.

Knuth-Bendix *completion* doesn't help.

- it doesn't terminate for many cases of practical interest, and
- the completed systems (e.g., for propositional logic) usually aren't useful in practice

"Inductionless induction" is impractical

- establishing the preconditions is usually as hard as doing regular induction

So we turned to a compatible and well-known logic.

First-order logic fits well with our use of ADT specifications in pre-/post-condition specifications of procedures.

Decision: Two-tiered Specification Strategy

We wanted to specify real programs

- in a variety of real programming languages
- we needed to be precise about the relation between
  - the programming language and
  - the specification language
  - What would it mean to implement a specification correctly?

Interface language tier–one per programming language

- types, state, parameter passing, flow of control, errors, . . .
  - *all the messy, boring stuff*
- based on particular PL's concepts and notation
- complexity proportionate to that of the PL

Decision: Two-tiered Specification Strategy, contd.

Shared Language tier

- *all the mathematically interesting stuff*
  - put all the specification subtlety in the simple language
- mathematical abstractions
- reusable specification components
- "putting together" operations
  - combine LSL specifications, not interface specifications
- independent of programming language(s)

Each interface language uses LSL specifications

- including a trait defining the base data types and memory model of its programming language

Decision:
Designed-in Redundancy for Checking Tools

Specifications are at least as error-prone as programs

- an unchecked specification is like an uncompiled program
- only one specification error per page in an already desk-checked specification is doing pretty well

It is important to check specifications incrementally

- reuse and extend existing specifications
- check partial specifications during development
- check specifications that are intentionally incomplete

We designed LSL with checking tools in mind

- and redesigned, as the tools evolved

Redundancy for Checking Tools

Mandatory declaration

Syntax and sort-checking (of course)

Consistency *

Assumptions

Optional claims

- intended degree of completeness of axioms
- intended implications of axioms

Much of this checking requires theorem-proving

- hence LP, the Larch Prover

We removed some claims that we didn't figure out how to check

- imports
- constrains

Decision: Proper Functions

All models must interpret operators as single-valued functions

- no "non-deterministic functions"

All models must interpret operators as total functions

- but operators may be underdetermined (models may be non-isomorphic)
- syntactically correct terms are never
  - "undefined"
  - "erroneous"
- this is a subtle source of surprises
  - but no worse than the alternatives
  - "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."
    —Jim Thatcher, 1977

Decisions: Types and Sorts

In each interface language, "type" must be consistent with what it means in its programming language.

In LSL, we need a more basic notion of "sort" that can be used by all interface languages.

- each type *based on* a sort

First try: same sort *iff* same identifier

- No subsorts
- No structured sorts
- Global scope for sort names

Second try: same sort *iff* same sort expression

- Still no subsorts
- Restricted structured sorts (e.g., Pair(Stack(Int), Bool))
- No sort equations
- Still global scope for sort names

Decisions: Types and Sorts, contd.

Very common idioms for

- enumerations
- tuples
- discriminated unions

Simple, but tedious to write

- non-linear number of axioms
- required invention of too many names
- even more tedious to read and recognize as an idiom

So we added special shorthands for these three idioms

- not expressable as traits
- haven't revisited this since adding sort expressions

Decisions: Syntax

Goals

- 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,
    $x \ \triangleleft\, x < y \,\triangleright\ y$
- Resolve operator overloading using the sorts of the operands.
- Allow expressions to be parsed without seeing declarations
  - because they may be in other traits

These goals are incompatible!

Decisions: Syntax, contd.

We devised a rather elaborate syntax for expressions.

- disambiguation depends on “correct” token classification
- but we already needed token customization to deal with PL lexical conventions

Problem: Renaming vs. Shorthands

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` expands to

```
introduces
  [__, __]: E, S -> C
  __.hd: C -> E
  __.tl: C -> E
  set_hd: C, E -> C
  set_tl: C, S -> C
asserts
  ...
```

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 utility of the shorthand.

Issue: Parameterized Operator Names?

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

"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 intergers, 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 using IntPairTrait(PairInt for IntPair) but that won't change the name equal_as_IntPair to be equal_as_PairInt."—Gary Leavens

Problem: Hidden .init Files

Lexical information is taken from .init files, which can, by reclassifying tokens, completely change the interpretation of a trait. This is dangerous and can be very confusing, since, like most configuration files, .init files tend to be ignored.

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

Issue: Arithmetic

"Can we agree on hardwiring in 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?"—Steve Garland

Issue: Conservative Extension

Originally, LSL had a construct to indicate which operators were being “defined” (or whose definitions were being extended) by a group of axioms. But we did not know exactly what this “should” mean, or how to check it.

LSL also distinguished between “including” another trait and “importing” it. The intuitive notion was that we did not want to accidentally change the meaning of, say, Boolean or Integer, by axioms in a trait that used them. But we discovered that we did not have a precise meaning to go with this intuition.

Issue: Conservative Extension, contd.

"We need to put some notion of conservative extension back into LSL. 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. Hence it's important to provide specifiers with a means for making claims (such as conservative extension) about consistency that introduce checkable proof obligations." —Steve Garland

"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 'conserves' out of LSL because we thought checking conservative extension was too hard; we need to put something like it back in because checking conservative extension is too important to ignore." —Steve Garland

Several languages have a "definitional" or "shell" construct that allows operator definitions to be introduced in such a way that it is guaranteed that they do not introduce contradictions or allow the deduction of new properties of existing operators. Can this fit neatly into the LSL framework? With what restrictions?

Issue: Operator Precedence

The least satisfying property of LSL is its syntax for terms. We worked very hard to accommodate many of the most common notations. But the result is both too complex and too restrictive. It's probably about the hardest thing to teach about the language. ("Why do I need parentheses *there*?")

"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."—Steve Garland

Issue: Operator Precedence, contd.

Luca Cardelli's report SRC-97 presents 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. 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 a much simpler LSL base grammar, plus Luca's extension mechanism to introduce syntax for, say, infix arithmetic operators, arrays, or sets and set constructors in the trait that gives the corresponding axioms would make it easy to write each trait in its "most natural" syntax. I'm not sure how well the extensions will compose when traits are combined (especially, operator precedence).

Issue: Introducing Subsorting 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.

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
- quantifier subsorting
- widening

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

Issue: Introducing Subsorting, contd.

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 associates it with the strongest valid theory for an operator with that symbol.

Issue: Introducing Subsorting, contd.

The details are not worked out. For example, perhaps we need the rule that if an operator has two applicable overloadings that differ only in that the range of one is a subsort of the range of the other, then the subsort applies.

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, and would undoubtedly make a real mess of matching and rewriting.

Issue: Total Functions

Our choice is quite conventional in algebra and logic, somewhat less so in computer science, where computability is an issue.

We have relegated the issue of partiality (and what to do about it) to the interface language level. But can we get away with it?

At one level, the answer is surely Yes, because we can replace partial functions with relations—at a hideous notational cost.

`1/0` isn't undefined, we just don't give you any axioms that let you prove that it is equal to some fraction with a non-zero denominator.

Issue: Total Functions, contd.

But there are pitfalls for the unwary 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 fails to generate these non-standard terms
- the verifier must take them account in proofs by induction

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 define auxiliary operators (with common signatures, but different properties) to simplify some complicated definitions. To avoid 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

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

Summary and Conclusions

Language design is hard

- and it's certainly not finished until *somebody else* implements your language, too

The hardest problems come from interactions of features, so

- keep everything as simple as possible as long as you can
- know what the tradeoffs are
- 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 languages

- they concern getting people to use them!