60

Debugging Larch Shared Language Specifications

Stephen J. Garland and John V. Guttag

July 4, 1990

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in l984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Debugging Larch Shared Language Specifications

Stephen J. Garland and John V. Guttag

July 4, 1990

John Guttag and Stephen Garland were supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, and by the National Science Foundation under grant CCR-8910848. Authors' address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. E-mail: garland@lcs.mit.edu, guttag@lcs.mit.edu

Abstract

The Larch family of specification languages supports a two-tiered definitional approach to specification. Each specification has components written in two languages: one designed for a specific programming language and another independent of any programming language. The former are called *Larch interface languages*, and the latter the *Larch Shared Language* (*LSL*).

The Larch style of specification emphasizes brevity and clarity rather than executability. To make it possible to test specifications without executing or implementing them, Larch permits specifiers to make claims about logical properties of specifications and to check these claims at specification time. Since these claims are undecidable in the general case, it is impossible to build a tool that will automatically certify claims about arbitrary specifications. However, it is feasible to build tools that assist specifiers in checking claims as they debug specifications. This paper describes the checkability designed into LSL and discusses two tools that help perform the checking.

This paper is a revised and expanded version of a paper presented at the April 1990 IFIP Working Conference on Programming Concepts and Methods [7].

Index terms. Formal specifications, Larch, programming, theorem proving, validation.

Contents

1 Introduction

Proponents of formal specifications argue that the susceptibility of formal specifications to machine analysis and manipulation increases their value and reduces their cost. The Larch project [9, 10, 11, 12] seeks to support this position by building and using tools that facilitate the construction of formal specifications for program modules.

It is not sufficient for specifications to be precise; they should also accurately reflect the specifier's intentions. Without accuracy, precision is useless and misleading. Mistakes from many sources will crop up in specifications. Any practical methodology that relies on specifications must provide means for detecting and correcting their flaws, in short, for debugging them. Parsing and type-checking are useful and easy to do, but don't go far enough. On the other hand, we cannot prove the "correctness" of a specification, because there is no absolute standard against which to judge correctness. So we seek tools that will be helpful in detecting and localizing the kinds of errors that we commonly observe.

The Larch style of specification emphasizes brevity and clarity rather than executability, so it is usually impossible to validate Larch specifications by testing. Instead, Larch allows specifiers to make precise claims about specifications—claims that, if true, can be verified at specification time. While verification cannot guarantee that a specification meets a specifier's intent, it is a powerful debugging technique; once we have removed the flaws it reveals, we have more confidence in the accuracy of a specification.

The claims allowed in Larch specifications are undecidable in the general case, so it is impossible to build a tool that will automatically certify an arbitrary specification. However, it is feasible to build tools that assist specifiers in checking claims as they debug specifications.

This paper describes how two such tools fit into our work on LSL, the Larch Shared Language. LP (the Larch Prover) is our principal debugging tool. Its design and development have been motivated primarily by our work on LSL, but it also has other uses (for example, reasoning about circuits and concurrent algorithms [6, 19]). Because of these other uses, and because we also intend to use LP to analyze Larch interface specifications, we have tried not to make LP too LSL-specific. Instead, we have chosen to build a second tool, LSLC (the LSL Checker), to serve as a front-end to LP. LSLC checks the syntax and static semantics of LSL specifications and generates LP proof obligations from their claims. These proof obligations fall into three categories: consistency (that a specification does not contradict itself), theory containment (that a specification has intended consequences), and relative completeness (that a set of operators is adequately defined).

Section 2 provides a brief introduction to Larch. Sections 3 and 4 describe the checkable claims that can be made in LSL specifications. Sections 5 through 8 describe how LP is used to check these claims. Section 9 contains an extended example illustrating how LP is used to debug LSL specifications. The concluding section summarizes the current state of our research and plans.

$addWindow =$ **proc** $(v : View, w : Window, c : Coord)$ **signals** $(duplicate)$
modifies v
ensures $v' = addW(v, w, c)$
except when $w \in v$ **signals** $duplicate$ **ensures** $v' = v$

Figure 1: Sample Larch/CLU Interface Specification

2 The Larch family of specification languages

The Larch family of specification languages supports a two-tiered definitional approach to specification [12]. Each specification has components written in two languages: one designed for a specific programming language and another independent of any programming language. The former are called *Larch interface languages*, and the latter the *Larch Shared Language* (*LSL*).

Larch interface languages are used to specify the interfaces between program components. Each specification provides the information needed to use the interface and to write programs that implement it. A critical part of each interface is how the component communicates with its environment. Communication mechanisms differ from programming language to programming language, sometimes in subtle ways. We have found it easier to be precise about communication when the interface specification language reflects the programming language. Specifications written in such interface languages are generally shorter than those written in a "universal" interface language. They are also clearer to programmers who implement components and to programmers who use them.

Each Larch interface language deals with what can be observed about the behavior of components written in a particular programming language. It incorporates programming-language-specific notations for features such as side effects, exception handling, iterators, and concurrency. Its simplicity or complexity depends largely upon the simplicity or complexity of the observable state and state transformations of its programming language. Figure 1 contains a sample interface specification for a CLU procedure in a window system.

Larch Shared Language specifications are used to provide a semantics for the primitive terms used in interface specifications. Specifiers are not limited to a fixed set of primitive terms, but can use LSL to define specialized vocabularies suitable for particular interface specifications. For example, an LSL specification would be used to define the meaning of the symbols $\in$ and *addW* in Figure 1, thereby precisely answering questions such as what it means for a window to be in a view (visible or possibly obscured?), or what it means to add a window to a view that may contain other windows at the same location.

The Larch approach encourages specifiers to keep most of the complexity of specifications in the LSL tier for several reasons:

- LSL abstractions are more likely to be re-usable than interface specifications.

$LinearContainer(E, C)$: **trait**

introduces

new: $\rightarrow C$

insert: $C, E \rightarrow C$

next: $C \rightarrow E$

rest: $C \rightarrow C$

isEmpty: $C \rightarrow Bool$

$\_\_ \in \_\_$: $E, C \rightarrow Bool$

asserts

C **generated by** *new*, *insert*

C **partitioned by** *next*, *rest*, *isEmpty*

forall c: C, e, e': E

$next(insert(new, e)) == e$

$rest(insert(new, e)) == new$

$isEmpty(new)$

$\neg isEmpty(insert(c, e))$

$\neg(e \in new)$

$e \in insert(c, e') == e = e' \mid e \in c$

implies

forall c: C, e: E

$isEmpty(c) \Rightarrow \neg(e \in c)$

converts $\in$, $isEmpty$

Figure 2: Sample LSL Specification

- LSL has a simpler underlying semantics than most programming languages (and hence than most interface languages), so that specifiers are less likely to make mistakes.
- It is easier to make and check claims about semantic properties of LSL specifications than about semantic properties of interface specifications.

This paper concentrates on the problem of debugging LSL specifications.

3 Semantic checks in the Larch Shared Language

A precise definition of the Larch Shared Language, including associated semantic checks, is contained in [9, 11]. This section informally describes LSL and these checks by considering claims that specifiers can make about some sample *traits*—LSL's basic units of specification.

A trait specifies the properties of a collection of operators. The trait *LinearContainer* in Figure 2, for example, specifies properties common to a number of abstract data types in which objects (of some sort C) contain elements (of some sort E) in a definite order. Types exhibiting these properties include stacks, queues, priority queues, sequences, and vectors. The trait can be used in specifying additional generic operators for these data types, and it can be specialized to specify particular data types.

The **asserts** clause associates a multisorted first-order theory with a trait. The axioms in the theory consist of the equations following the quantifier **forall** in the **asserts** clause (equations of the form $t == true$ are abbreviated to t), an induction scheme associated with each **generated by**, and an axiom associated with each **partitioned by**. For example, the theory of *LinearContainer* is axiomatized by its six equations, an axiom schema obtained from the **generated by**,

$$(\phi(new) \wedge (\forall c : C)(\forall e : E)\,[\phi(c) \Rightarrow \phi(insert(c, e))]\,) \;\Rightarrow\; (\forall c' : C)\phi(c'),$$

which can be instantiated with any first-order formula ϕ, and an axiom obtained from the **partitioned by**,

$$(\forall c, c' : C)\,(\,[\;(isEmpty(c) \Leftrightarrow isEmpty(c')) \wedge (next(c) = next(c')) \wedge (rest(c) = rest(c'))\,\big] \;\Rightarrow\; c = c'\big)$$

The theories of LSL traits are closed under logical consequence.

The semantics of LSL is based on first-order logic rather than on initial or final algebras, for two reasons. First, it is important that we be able to construct and reason about specifications incrementally. By treating the assertions of a trait as axioms in a first-order theory, we ensure that adding assertions (even about new operators) to a trait does not remove facts from its associated theory. Because the initial and final algebra interpretations of sets of equations are not monotonic in this sense, they provide less suitable semantics than does first-order logic. Second, the **generated by** and **partitioned by** constructs in LSL, which have natural interpretations in first-order logic, provide greater flexibility for axiomatizing traits than do the initial or final algebra interpretations.

Semantic claims in LSL

Semantic claims about LSL traits fall into three categories: consistency, theory containment, and relative completeness. Consistency is always required, that is to say, no LSL theory should contain the inconsistent equation $true == false$. Claims in the other two categories are made by specific LSL constructs.

The **implies** clause adds nothing to the theory of a trait. Instead, it makes a claim about theory containment. It enables specifiers to include information they believe to be redundant, either as a check on their understanding or to call attention to something that a reader might otherwise miss. The redundant information is of two kinds: statements like those in **asserts** clauses and **converts** clauses that describe the extent to which a specification is claimed to be complete.

The initial design of LSL incorporated a built-in notion of completeness. We quickly concluded, however, that requirements of completeness are better left to the specifier's discretion. Not only is it useful to check certain aspects of completeness long before a

$PriorityQueue(E, Q)$: **trait**
  assumes $TotalOrder(E)$
  includes $LinearContainer(E, Q)$
  asserts forall q: Q, e: E
    $next(insert(q, e)) ==$
      if $isEmpty(q)$ **then** e
      else if $next(q) < e$ **then** $next(q)$ **else** e
    $rest(insert(q, e)) ==$
      if $isEmpty(q)$ **then** new
      else if $next(q) < e$ **then** $insert(rest(q), e)$ **else** q
  implies
    forall q: Q, e: E
      $e \in q \Rightarrow \neg(e < next(q))$
    converts $next$, $rest$, $isEmpty$, $\in$ **exempting** $next(new)$, $rest(new)$

Figure 3: LSL Specification for a Priority Queue

specification is finished, but most finished specifications are left intentionally incomplete in places. LSL allows specifiers to make checkable claims about how complete they intend specifications to be. These claims are usually more valuable when a specification is revised than when it is first written. Specifiers don't usually make erroneous claims about completeness when first writing a specification. On the other hand, when editing a specification, they frequently delete or change something without realizing its impact on completeness.

The **converts** clause in *LinearContainer* claims that the trait contains enough axioms to define $\in$ and *isEmpty*. Exactly what it means to "define" an operator was one of the more delicate design issues for LSL and has been changed in the most recent version of the language. This **converts** clause claims that, given any fixed interpretations for the other operators, all interpretations of $\in$ and *isEmpty* that satisfy the trait's axioms are the same.

The **converts** clause in *PriorityQueue* (Figure 3) involves more subtle checking. The **exempting** clause indicates that the lack of equations for $next(new)$ and $rest(new)$ is intentional: the operators *next* and *rest* are only claimed to be defined uniquely relative to interpretations for the terms $next(new)$ and $rest(new)$. Section 7 describes the checking entailed by the **converts** clause in more detail.

Checking composed LSL specifications

There are two mechanisms for combining LSL specifications. Both are defined as operations on the texts of specifications, rather than on theories or models [3, 17]. For both mechanisms, the theory of a combined specification is axiomatized by the union of the axiomatizations for the individual specifications; each operator is constrained by the axioms of all traits in which it appears. Trait inclusion and trait assumption differ only in the checking they entail.

$TotalOrder(E)$: **trait**
  introduces
    $\_\_ < \_\_$: $E, E \rightarrow Bool$
    $\_\_ > \_\_$: $E, E \rightarrow Bool$
  asserts forall x, y, z: E
    $\neg(x < x)$
    $(x < y \;\&\; y < z) \Rightarrow x < z$
    $x < y \mid x = y \mid y < x$
    $x > y == y < x$
  implies
    $TotalOrder(E, > \; for \; <, < \; for \; >)$
    forall x, y: E $\neg(x < y \;\&\; y < x)$

Figure 4: LSL Specification for Total Orders

The trait *PriorityQueue*, which includes *LinearContainer*, further constrains the interpretations of *next*, *rest*, and *insert*. Its **assumes** clause indicates that its theory also contains that of the trait *TotalOrder* shown in Figure 4. The use of **assumes** rather than **includes** entails additional checking, namely that the assumption must be discharged whenever *PriorityQueue* is incorporated into another trait. For example, checking the trait

NatPriorityQueue: **trait**
  includes *PriorityQueue*(*Nat*, *NatQ*), *NaturalNumber*

involves checking that the assertions in the traits *PriorityQueue*, *LinearContainer*, and *NaturalNumber* together imply those of *TotalOrder*(*Nat*).

Figure 5 summarizes the checking (beyond consistency) that LSL requires for the sample traits introduced in this section.

4 Proof obligations for LSL specifications

An LSL specification generally consists of a hierarchy of traits, some of which include, assume, or imply others. We use the LSL Checker (LSLC) to check the syntax and static semantics of the traits, to formulate the proof obligations required to check the semantic claims in the traits, and to discharge some of these proof obligations. This section describes how LSLC extracts the proof obligations. The next several sections describe how we use LP to discharge those proof obligations that LSLC cannot discharge "by inspection."

As LSLC extracts proof obligations from a specification, it checks for cycles in the trait hierarchy. Let $\sqsubset^+$ be the transitive closure of the relation defined by setting $S \sqsubset T$ iff T includes or assumes S. Let $\Rightarrow^+$ be the transitive closure of the relation defined by setting $S \Rightarrow T$ iff S implies T. Then LSLC checks the following two conditions.

NatPriorityQueue
Check assumption of $TotalOrder(Nat)$ by *PriorityQueue*. Use the assertions of all traits except *TotalOrder*.

PriorityQueue
Check implications. Use the assertions of *PriorityQueue* and the theories of *LinearContainer* and *TotalOrder*.

NaturalNumber
Check . . . Use . . .

LinearContainer
Check implications. Use local assertions.

TotalOrder
Check implications. Use local assertions.

Figure 5: Summary of Required Checking

Condition 1: $\sqsubset^+$ is a strict partial order.

Condition 2: There are no traits S and T such that both $S \sqsubset^+ T$ and $S \Rightarrow^+ T$.

These conditions ensure that the traits can be checked separately. The soundness of separate checks is shown by induction on the trait hierarchy defined by Condition 1, which must be satisfied. If Condition 2 is also satisfied, then the implications of traits below a trait T in the hierarchy can safely be used when checking T.

Note that $\Rightarrow^+$ need not be a strict partial order. In some specifications, we may want to assert that two traits S and T are equivalent, that is to say, that S implies T and T implies S. It may even be the case that S and T are the same trait, as in *TotalOrder* (see Figure 4), where we wish to assert that $>$, like $<$, is a total ordering relation.

LSLC extracts six sets of propositions (equations, **generated by** clauses, and **partitioned by** clauses) from each trait T in a trait hierarchy, as follows.

- The *assertions* of T consist of the propositions in the **asserts** clauses of T and of all traits (transitively) included in T.
- The *assumptions* of T consist of the assertions of all traits (transitively) assumed by T.
- The *axioms* of T consist of its assertions and its assumptions.
- The *immediate consequences* of T consist of the propositions in T's **implies** clause and the axioms of all traits that T explicitly implies.
- The *explicit theory* of T consists of its axioms, the propositions in its **implies** clause, and the explicit theories of all traits S such that $S \sqsubset^+ T$ or $T \Rightarrow^+ S$. (The explicit theory of T, unlike the theory of T as defined in Section 3, is a finite set and is not closed under logical consequence.)

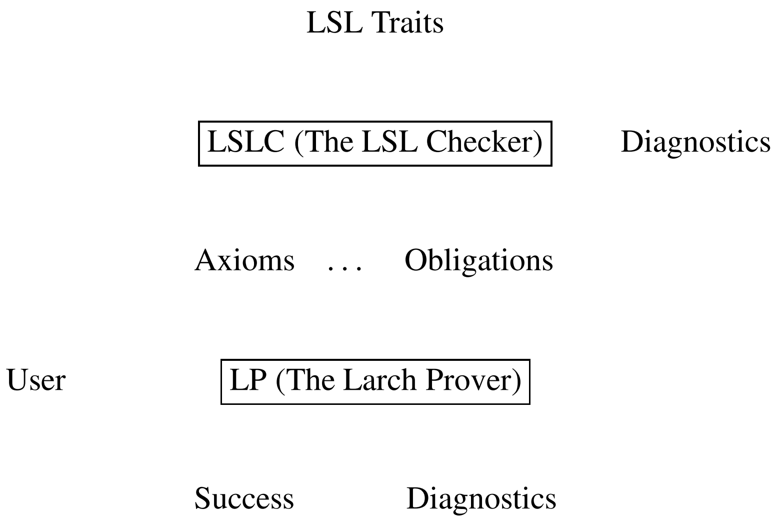

Figure 6: Using LSLC and LP to Check LSL Traits

- The *lemmas* available for checking T, when Condition 2 is satisfied, consist of the explicit theories of all traits S such that $S \sqsubset^+ T$.

To check a hierarchy of traits, we must prove that the axioms of each trait T are consistent, and we must discharge the following proof obligations:

- T's immediate consequences must follow from its axioms. If Condition 2 is satisfied, it is sound to use the lemmas available for T when performing this check.
- T's **converts** clauses must follow from its explicit theory. (The preceding proof obligation ensures that T's explicit theory follows from its axioms.)
- The assumptions of each trait explicitly included in T must follow from T's axioms.

LSLC can discharge some proof obligations "by inspection," for example, because a proposition to be proved occurs textually among the facts available for use in the proof. At other times, LSLC must formulate commands for LP that initiate a proof of the proposition. Sometimes LP will be able to carry out the required proof automatically; sometimes it will require user assistance. Figure 6 shows how LSLC and LP can be used together to check LSL traits.

Consider, for example, the traits *PriorityQueue*, which assumes *TotalOrder*, and *NatPriorityQueue*, which includes both *PriorityQueue* and *NaturalNumber*. If *NaturalNumber* explicitly includes or implies *TotalOrder*, or if the assertions of *TotalOrder* are among the axioms of *NaturalNumber*, then LSLC can discharge the assumption required for including *PriorityQueue* in *NatPriorityQueue*. On the other hand, if *NaturalNumber* simply asserts some properties of the binary relations $<$ and $>$, then LSLC will formulate LP commands that initiate a proof of the conjecture that these properties imply the assertions of *TotalOrder*.

By providing a small set of axioms for a trait T, a specifier can make it easier to check traits that imply T or that include a trait that assumes T. By providing a large set of implications for T, a specifier can make it easier to reason about T and, in particular, to check traits that include or assume T, without at the same time making it harder to check traits that imply T or that include a trait that assumes T.

5 Translating LSL traits into LP

The basis for proofs in LP is a logical system. This section contains an overview of the components of a logical system in LP and discusses their relation to the components of an LSL trait. The following sections discuss how these components are used by LP to discharge proof obligations associated with LSL traits.

A logical system in LP consists of a signature (given by declarations) and equations, rewrite rules, operator theories, induction rules, and deduction rules (all expressed in a multisorted fragment of first-order logic). These systems are closely related to LSL theories, but are handled in somewhat different ways, both because axioms in LP have operational content as well as semantic content and because they can be presented to LP incrementally, rather than all at once.

Declarations

Sorts, operators, and variables play the same roles in LP as they do in LSL. As in LSL, operators and variables must be declared, and operators can be overloaded. There are a few minor differences: Sorts must be declared in LP and LP doesn't provide scoping for variables. LP's syntax for terms is not yet as rich as LSL's, but we plan to rectify that; this paper uses LSL's term syntax throughout.

LSLC produces the LP declarations shown in Figure 7 from the **introduces** and **forall** clauses in the trait *LinearContainer*.

Equations and rewrite rules

Like LSL, LP is based on a fragment of first-order logic in which equations play a prominent role. Some of LP's inference mechanisms work directly with equations. Most, however, require that equations be oriented into rewrite rules, which LP uses to reduce terms to normal forms. It is usually essential that the rewriting relation be terminating, that is no term can be rewritten infinitely many times. LP provides several mechanisms that automatically orient many sets of equations into terminating rewriting systems. For example, in response to the commands

declare sort G

declare sorts E, C
declare variables c, c': C, e, e': E
declare operators
new: $\rightarrow C$
$insert$: $C, E \rightarrow C$
$next$: $C \rightarrow E$
$rest$: $C \rightarrow C$
$isEmpty$: $C \rightarrow Bool$
$\_\_ \in \_\_$: $E, C \rightarrow Bool$
..

Figure 7: LP Declarations Produced by LSLC from *LinearContainer*

declare variables x, y, z: G
declare operators e: $\rightarrow G$, i: $G \rightarrow G$, $\_\_ * \_\_$: $G, G \rightarrow G$
assert
$(x * y) * z == x * (y * z)$
$e == x * i(x)$
$e * x == x$
..

that enter the usual first-order axioms for groups, LP produces the rewrite rules

$$(x * y) * z \rightarrow x * (y * z)$$
$$x * i(x) \rightarrow e$$
$$e * x \rightarrow x$$

It automatically reverses the second equation to prevent nonterminating rewriting sequences such as $e \rightarrow e * i(e) \rightarrow i(e) \rightarrow i(e * i(e)) \rightarrow i(i(e)) \rightarrow \ldots$ The discussion of operator theories below treats the issue of termination further.

A system's rewriting theory (that is to say, the propositions that can be proved by reduction to normal form) is always a subset of its equational theory (that is to say, the propositions that follow logically from its equations and from its rewrite rules considered as equations). The proof mechanisms discussed in Section 6 compensate for the incompleteness that results when, as is usually the case, a system's rewriting theory does not include all of its equational theory. In the case of group theory, for example, the equation $e == i(e)$ follows logically from the second and third axioms, but is not in the rewriting theory of the three rewrite rules (because it is irreducible and yet is not an identity).

LP provides built-in rewrite rules to simplify terms involving the Boolean operators $\neg$, $\&$, $|$, $\Rightarrow$, and $\Leftrightarrow$, the equality operator $=$, and the conditional operator **if**. These rewrite rules are sufficient to prove many, but not all, identities involving these operators. Unfortunately, the sets of rewrite rules that are known to be complete for propositional calculus require exponential time and space. Furthermore, they can expand, rather than simplify, propositions that do not reduce to identities. These are serious drawbacks, because when we are debugging specifications we often attempt to prove conjectures that are not true. So none of the complete sets of rewrite rules is built into LP. Instead, LP provides proof mechanisms that can be used to overcome incompleteness in a rewriting system, and it allows users to add any of the complete sets they choose to use.

LP treats the equations $true == false$ and $x == t$, where t is a term not containing the variable x, as inconsistent. Inconsistencies can be used to establish subgoals in proofs by cases and contradiction. If they arise in other situations, they indicate that the axioms in the logical system are inconsistent.

Operator theories

LP provides special mechanisms for handling equations such as $x + y == y + x$ that cannot be oriented into terminating rewrite rules. The LP command **assert ac** $+$ says that $+$ is associative and commutative. Logically, this assertion is merely an abbreviation for two equations. Operationally, LP uses it to match and unify terms modulo associativity and commutativity. This not only increases the number of theories that LP can reason about, but also reduces the number of axioms required to describe various theories, the number of reductions necessary to derive identities, and the need for certain kinds of user interaction, for example, case analysis. The main drawback of term rewriting modulo operator theories is that it can be much slower than conventional term rewriting.

LP recognizes two nonempty operator theories: the associative-commutative theory and the commutative theory. It contains a mechanism (based on user-supplied polynomial interpretations of operators) for ordering equations that contain commutative and associative-commutative operators into terminating systems of rewrite rules. But this mechanism is difficult to use, and most users rely on simpler ordering methods based on LP-suggested partial orderings of operators. These simpler ordering methods do not guarantee termination when equations contain commutative or associative-commutative operators, but they work well in practice. Like manual ordering methods, which give users complete control over whether equations are ordered from left to right or from right to left, they are easy to use. In striking contrast to manual ordering methods, they have not yet caused difficulties by producing a nonterminating set of rewrite rules.

Induction rules

LP uses induction rules to generate subgoals to be proved for the basis and induction steps in proofs by induction. The syntax for induction rules is the same in LP as in LSL.[1] Users can specify multiple induction rules for a single sort, for example, by the LP commands

[1]The semantics of induction is stronger in LSL than in LP, where arbitrary first-order formulas cannot be written.

declare sorts E, S
declare operators
$\{\}$: $\rightarrow S$
$\{\_\_\}$: $E \rightarrow S$
$\_\_ \cup \_\_$: $S, S \rightarrow S$
insert: $S, E \rightarrow S$
..
set name *setInduction1*
assert S **generated by** {}, *insert*
set name *setInduction2*
assert S **generated by** {}, {__}, ∪

and can use the appropriate rule when attempting to prove an equation by induction; for example,

prove $x \subseteq (x \cup y)$ **by induction on** x **using** *setInduction2*

In LSL, the axioms of a trait typically have only one **generated by** for a sort. It is often useful, however, to put others in the trait's implications.

Deduction rules

LP subsumes the logical power of the **partitioned by** construct of LSL in deduction rules, which LP uses to deduce equations from other equations and rewrite rules. Like other formulas in LP, deduction rules may be asserted as axioms or proved as theorems. While the **partitioned by** in the trait *LinearContainer* can be expressed by an equation, in general a **partitioned by** is equivalent to a universal-existential axiom, which can be expressed as a deduction rule in LP. For example, the LP commands

declare sorts E, S
declare operator $\in$: $E, S \rightarrow Bool$
declare variables e: E, x, y: S
assert when $(forall\ e)\ e \in x == e \in y$ **yield** $x == y$

define a deduction rule equivalent to the axiom

$$(\forall x, y : S)\,[(\forall e : E)(e \in x \Leftrightarrow e \in y) \Rightarrow x = y]$$

of set extensionality, which can also be expressed by **assert** S **partitioned by** $\in$ in LP, as in LSL. This deduction rule enables LP to deduce equations such as $x == x \cup x$ automatically from equations such as $e \in x == e \in (x \cup x)$.

Deduction rules also serve to improve the performance of LP and to reduce the need for user interaction. Examples of such deduction rules are the built-in &-splitting law

declare variables p, q: $Bool$
when $p \mathbin{\&} q == true$ **yield** $p == true, q == true$

and the cancellation law for addition

declare variables x, y, z: Nat
when $x + y == x + z$ **yield** $y == z$

LP automatically applies deduction rules to equations and rewrite rules whenever they are normalized. The sample proof in Section 7 illustrates the logical power of deduction rules, as well as the benefits of applying them automatically to the case and induction hypotheses in a proof.

6 Proof mechanisms in LP

This section provides a brief overview of the proof mechanisms in LP. The next two sections discuss how they are used to check semantic claims about LSL specifications.

LP provides mechanisms for proving theorems using both forward and backward inference. Forward inferences produce consequences from a logical system; backward inferences produce lemmas whose proof will suffice to establish a conjecture. There are four methods of forward inference in LP.

- Automatic *normalization* produces new consequences when a rewrite rule is added to a system. LP keeps rewrite rules, equations, and deduction rules in normal form. If an equation or rewrite rule normalizes to an identity, it is discarded. If the hypothesis of a deduction rule normalizes to an identity, the deduction rule is replaced by the equations in its conclusions. Users can "immunize" equations, rewrite rules, and deduction rules to protect them from automatic normalization, both to enhance the performance of LP and to preserve a particular form for use in a proof. Users can also "deactivate" rewrite rules and deduction rules to prevent them from being automatically applied.

- Automatic *application of deduction rules* produces new consequences after equations and rewrite rules in a system are normalized. Deduction rules can also be applied explicitly, for example, to immune equations.

- The computation of *critical pairs* and the Knuth-Bendix *completion* procedure [13, 16] produce consequences (such as $i(e) == e$) from incomplete rewriting systems (such as the three rewrite rules for groups). We rarely complete our rewriting systems, for the reasons discussed in [5]. However, we often make selective use of critical pairs. As discussed in Section 9, we also use the completion procedure to look for inconsistencies.

- Explicit *instantiation* of variables in equations, rewrite rules, and deduction rules also produces consequences. For example, in a system that contains the rewrite rules $a < (b + c) \rightarrow true$ and $(b + c) < d \rightarrow true$, instantiating the deduction rule

$$\textbf{when } x < y == true,\ y < z == true \textbf{ yield } x < z == true$$

with a for x, $b+c$ for y, and d for z produces a deduction rule whose hypotheses normalize to identities, thereby yielding the conclusion $a < d \rightarrow true$.

There are also six methods of backward inference for proving equations in LP. These methods are invoked by the **prove** command. In each method, LP generates a set of subgoals to be proved, that is, lemmas that together are sufficient to imply the conjecture. For some methods, it also generates additional axioms that may be used to prove particular subgoals.

- *Normalization* rewrites conjectures. If a conjecture normalizes to an identity, it is a theorem. Otherwise the normalized conjecture becomes the subgoal to be proved.
- *Proofs by cases* can further rewrite a conjecture. The command **prove** e **by cases** $t_1, \ldots, t_n$ directs LP to prove an equation e by division into cases $t_1, \ldots, t_n$ (or into two cases, t_1 and $\neg t_1$, if $n = 1$). One subgoal is to prove $t_1 \mid \ldots \mid t_n$. In addition, for each i from 1 to n, LP substitutes new constants for the variables of t_i in both t_i and e to form t_i' and e_i', and it creates a subgoal e_i' with the additional hypothesis $t_i' \rightarrow true$. If an inconsistency results from adding the case hypothesis t_i', that case is impossible, so e_i' is vacuously true.

  Case analysis has two primary uses. If the conjecture is a theorem, a proof by cases may circumvent a lack of completeness in the rewrite rules. If the conjecture is not a theorem, an attempted proof by cases may simplify the conjecture and make it easier to understand why the proof is not succeeding.
- *Proofs by induction* are based on the induction rules described in Section 5.
- *Proofs by contradiction* provide an indirect method of proof. If an inconsistency follows from adding the negation of the conjecture to LP's logical system, then the conjecture is a theorem.
- *Proofs of implications* can be carried out using a simplified proof by cases. The command **prove** $t_1 \Rightarrow t_2$ **by** $\Rightarrow$ directs LP to prove the subgoal t_2' using the hypothesis $t_1' \rightarrow true$, where t_1' and t_2' are obtained as in a proof by cases. (This suffices because the implication is vacuously true when t_1' is false.)
- *Proofs of conjunctions* provide a way to reduce the expense of equational term rewriting. The command **prove** t_1 & $\ldots$ & t_n **by** & directs LP to prove $t_1, \ldots, t_n$ as subgoals.

LP allows users to determine which of these methods of backward inference are applied automatically and in what order. The LP command

set proof-method & , $\Rightarrow$, **normalization**

directs LP to use the first of the three named methods that applies to a given conjecture.

set name *TotalOrder*
declare sort E
declare variables x, y, z: E
declare operators $\_\_ < \_\_, \_\_ > \_\_$: $E, E \rightarrow Bool$
assert
$\neg(x < x)$
$(x < y \; \& \; y < z) \Rightarrow x < z$
$x < y \mid x = y \mid y < x$
$x > y == y < x$
..

Figure 8: The File *TotalOrder_Assertions.lp*

Proofs of interesting conjectures hardly ever succeed on the first try. Sometimes the conjecture is wrong. Sometimes the formalization is incorrect or incomplete. Sometimes the proof strategy is flawed or not detailed enough. When an attempted proof fails, we use a variety of LP facilities (for example, case analysis) to try to understand the problem. Because many proof attempts fail, LP is designed to fail relatively quickly and to provide useful information when it does. It is not designed to find difficult proofs automatically. Unlike the Boyer-Moore prover [1], it does not perform heuristic searches for a proof. Unlike LCF [15], it does not allow users to define complicated search tactics. Strategic decisions, such as when to try induction, must be supplied as explicit LP commands (either by the user or by a front-end such as LSLC). On the other hand, LP is more than a "proof checker," since it does not require proofs to be described in minute detail. In many respects, LP is best described as a "proof debugger."

7 Checking theory containment

The proof obligations triggered by **implies** and **assumes** clauses in an LSL trait require us to check theory containment, that is to check that claims follow from axioms. This section discusses how LSLC formulates the proof obligations for theory containment for LP, as well as how we use LP to discharge these obligations. The next section discusses checking consistency.

Proving an equation

The proof obligation for an equation is easy to formulate. Consider, for example, the proof obligations that must be discharged to check the trait *TotalOrder* shown in Figure 4. Figure 8 displays the LP commands that LSLC extracts from this trait in order to axiomatize its theory, and Figure 9 displays the LP commands that LSLC extracts in order to discharge

execute *TotalOrder_Assertions*
set name *TotalOrder_Consequences*
% Prove implication of *TotalOrder* with $>$ for $<$, $<$ for $>$
prove $\neg(x > x)$
  qed
prove $(x > y \,\&\, y > z) \Rightarrow x > z$
  qed
prove $x > y \mid x = y \mid y > x$
  qed
prove $x < y == y > x$
  qed
% Prove implied equation
prove $\neg(x > y \,\&\, y > x)$
  qed

Figure 9: The File *TotalOrder_Obligations.lp*

its proof obligations. The **execute** command obtains the axioms for *TotalOrder* from the file *TotalOrder_Assertions.lp*. The **prove** commands initiate proofs of the five immediate consequences of *TotalOrder*. LP can discharge all proof obligations except the last without user assistance. The user is alerted to the need to supply assistance in the last proof by a diagnostic ("incomplete proof") printed in response to the last **qed** command. At this point, the user can complete the proof by entering the commands

critical-pairs *TotalOrder* **with** *TotalOrder*
qed

Proofs of equations require varying amounts of assistance. For example, when checking that *LinearContainer* implies $isEmpty(c) \Rightarrow \neg(e \in c)$, the single LP command

resume by induction on c

suffices to finish the proof.

When checking that *PriorityQueue* implies

$$e \in q \Rightarrow \neg(e < next(q)),$$

more guidance is required. This proof also proceeds by induction. LP proves the basis subgoal without assistance. For the induction subgoal, LP introduces a new constant q_0 to take the place of the universally-quantified variable q, adds

$$e \in q_0 \Rightarrow \neg(e < next(q_0))$$

as the induction hypothesis, and attempts to prove

$$e \in insert(q_0, v) \Rightarrow \neg(e < next(insert(q_0, v))),$$

prove $e \in q \Rightarrow \neg(e < next(q))$ **by induction on** q
resume by case $isEmpty(q_0)$
  % Handle case $isEmpty(q_0)$
  critical-pairs *caseHyp* **with** *LinearContainer*
  % Handle case $\neg isEmpty(q_0)$
  resume by case $v_0 < next(c_0)$
    % Handle case $v_0 < next(c_0)$
    resume by case $e_0 = v_0$
      % Case $e_0 = v_0$ succeeds
      % Handle case $e_0 \neq v_0$
      complete
    % Handle case $\neg(v_0 < next(c_0))$
    resume by case $e_0 = v_0$
      % Case $e_0 = v_0$ succeeds
      % Handle case $e_0 \neq v_0$
      critical-pairs *impliesHyp* **with** *inductHyp*

Figure 10: Guidance for Proof of *PriorityQueue* Implication

which reduces to

$$(e = v \mid e \in q_0) \Rightarrow$$
$$\neg(e < (if\ isEmpty(q_0)\ then\ v\ else\ if\ v < next(q_0)\ then\ v\ else\ next(q_0)))$$

LP now assumes the hypothesis of the implication, introducing new constants e_0 and v_0 to take the place of the variables e and v, and attempts to prove the conclusion of the implication from this hypothesis. At this point, further guidance is required. The command

resume by case $isEmpty(q_0)$

directs LP to divide the proof into two cases based on the boolean expression in the first **if**. In the first case, $isEmpty(q_0)$, the conclusion reduces to $\neg(e_0 < v_0)$. The command

critical-pairs $caseHyp\ with\ LinearContainer$

leads LP to deduce $\neg(e \in q_0)$ from the first implication of *LinearContainer* (which is available for use in the proof because *LinearContainer* is below *PriorityQueue* in the trait hierarchy). With this fact, LP is able to finish the proof in the first case automatically: it applies a built-in deduction rule to conclude $e \in q_0 \rightarrow false$, uses this rewrite rule and another built-in deduction rule to derive $e_0 \rightarrow v_0$ from the hypothesis of the implication, and uses this rewrite rule to help simplify the conclusion of the implication to an identity. The second case, $\neg isEmpty(q_0)$ requires more user assistance. Figure 10 shows the entire proof script.

Proving a "partitioned by"

Proving a **partitioned by** clause amounts to proving the validity of the associated deduction rule in LP. For example, LSLC formulates for LP the proof obligations associated with the **partitioned by** in the **implies** clause of Figure 11 as

> **execute** *Set_Assertions*
> **prove** S **partitioned by** $\subseteq$

and LP translates the **partitioned by** into the deduction rule

$$when\ (forall\ z)\ \ x \subseteq z == y \subseteq z,\ \ z \subseteq x == z \subseteq y\ yield\ x == y$$

(This deduction rule contains two hypotheses because $\subseteq$ is a binary operator; either hypothesis is sufficient, but at present there is no way to indicate this in LSL or LP.)

LP generates a subgoal for proving the deduction rule by introducing two new constants, x_0 and y_0 of sort S, assuming $x_0 \subseteq z == y_0 \subseteq z$ and $z \subseteq x_0 == z \subseteq y_0$ as additional hypotheses, and attempting to prove the subgoal $x_0 == y_0$. LP cannot prove $x_0 == y_0$ directly because the equation is irreducible. The user can guide the proof by typing the command

> **instantiate** z **by** $\{e\}$ **in** *whenHyp*

which yields the lemma $e \in x_0 == e \in y_0$, after which LP automatically completes the proof by applying the deduction rule associated with the assertion S **partitioned by** $\in$.

Proving a "generated by"

Proving a **generated by** clause involves proving that the set of elements generated by the given operators contains all elements of the sort. For example, LSLC formulates for LP the proof obligation associated with the **generated by** in the **implies** clause of Figure 11 as

> **execute** *Set_Assertions*
> **prove** S **generated by** $\{\}$, $\{\_\_\}$, $\cup$

LP then introduces a new operator $isGenerated : S \rightarrow Bool$, adds the hypotheses

$$isGenerated(\{\})$$
$$isGenerated(\{e\})$$
$$(isGenerated(x)\ \&\ isGenerated(y)) \Rightarrow isGenerated(x \cup y)$$

and attempts to prove the subgoal $isGenerated(x)$. User guidance is required to complete the proof, for example, by entering the commands

> **resume by induction** x
> **complete**

$Set(E)$: **trait**

introduces

$\{\}$: $\to S$

insert: $S, E \to S$

$\{\_\_\}$: $E \to S$

$\_\_ \cup \_\_$: $S, S \to S$

$\_\_ \in \_\_$: $E, S \to Bool$

$\_\_ \subseteq \_\_$: $S, S \to Bool$

asserts

S **generated by** $\{\}$, *insert*

S **partitioned by** $\in$

forall s, s': S, e, e': E

$\{e\} == insert(\{\}, e)$

$s \cup \{\} == s$

$s \cup insert(s', e) == insert(s \cup s', e)$

$\neg(e \in \{\})$

$e \in insert(s, e') == e \in s \mid e = e'$

$\{\} \subseteq s$

$insert(s, e) \subseteq s' == s \subseteq s' \ \& \ e \in s'$

implies

S **generated by** $\{\}$, $\{\_\_\}$, $\cup$

S **partitioned by** $\subseteq$

forall s, s', s'': S, e, e': E

$e \in \{e'\} == e = e'$

$insert(s, e) == s \cup \{e\}$

$insert(insert(s, e), e') == insert(insert(s, e'), e)$

$\neg(insert(s, e) = \{\})$

$e \in (s \cup s') == e \in s \mid e \in s'$

$(s' \cup s'') \subseteq s == s' \subseteq s \ \& \ s'' \subseteq s$

converts $\{\_\_\}$, $\cup$, $\in$, $\subseteq$

converts *insert*, $\in$, $\subseteq$

Figure 11: An LSL Trait for Finite Sets

directing LP to use the induction scheme obtained from the assertion

S **generated by** {}, *insert*

and then to run the completion procedure to draw the necessary inferences from the additional hypotheses.

Proving a "converts"

Proving that a trait converts a set of operators involves showing that the axioms of the trait define the operators in the set relative to the other operators in the trait. For example, to show that *LinearContainer* converts *isEmpty* and $\in$, we must show that, given any interpretations for {} and *insert*, there are unique interpretations for *isEmpty* and $\in$ that satisfy the assertions of *LinearContainer*. Equivalently, we must show that the theory of *LinearContainer* together with the theory of *LinearContainer*($isEmpty'$ **for** $isEmpty$, $\in'$ **for** $\in$) imply the equations $isEmpty'(c) == isEmpty(c)$ and $e \in' c == e \in c$.

LSLC formulates this proof obligation for LP by making two copies of the file *LinearContainer.lp*, which contains the explicit theory of *LinearContainer*; in the second copy, *LinearContainer_Converts.lp*, all occurrences of *isEmpty* and $\in$ are replaced by $isEmpty'$ and $\in'$. By producing these two files, LSLC can formulate the proof obligation with the following LP commands:

execute *LinearContainer*
execute *LinearContainer_Converts*
prove $isEmpty'(c) == isEmpty(c)$
  qed
prove $e \in' c == e \in c$
  qed

The only user guidance required to prove this **converts** clause is a command to proceed by induction on c.

The proof obligation for the **converts** clause in *PriorityQueue* is similar. Here we must show that given any interpretations for {} and *insert*, as well as for the exempted terms $next(new)$ and $rest(new)$, there are unique interpretations for *next*, *rest*, *isEmpty*, and $\in$ that satisfy the assertions of *PriorityQueue*. As before, LSLC formulates this proof obligation for LP by making the required copy of *PriorityQueue.lp* and by generating the following LP commands:

execute *PriorityQueue*
execute *PriorityQueue_Converts*
assert $next'(new) == next(new)$
assert $rest'(new) == rest(new)$
prove $next'(q) == next'(q)$

qed

prove $rest'(q) == rest(q)$

qed

prove $isEmpty'(q) == isEmpty(q)$

qed

prove $e \in' q == e \in q$

qed

Again, the only user guidance required to complete the proofs are commands to proceed by induction on q.

8 Checking consistency

Checks for theory containment fall into the typical pattern of use of a theorem prover. The check for consistency is harder to formulate because it involves nonconsequence rather than consequence. We are still evaluating several approaches to this check. It seems probable that techniques for detecting when this check fails will be more useful than techniques for certifying that it succeeds.

A standard approach in logic to proving consistency involves interpreting the theory being checked in another theory whose consistency is assumed (for example, Peano arithmetic) or has been established previously [18]. In this approach, user assistance is required to define the interpretation. The proof that the interpretation satisfies the axioms of the trait being checked then becomes a problem of showing theory containment, for which LP is well suited. This approach is cumbersome and unattractive in practice. More promising approaches are based on metatheorems in first-order logic that can be used for restricted classes of specifications. For example, any extension by definitions (see [18]) of a consistent theory is consistent.

For equational traits (that is to say, traits with purely equational axiomatizations, of which there are relatively few), questions about consistency can be translated into questions about critical pairs. In some cases, we can use LP to answer these questions by running the completion procedure or by computing critical pairs. If these actions generate an inconsistency, the axioms are inconsistent; if they complete the axioms without generating the equation $true == false$, then the trait is consistent. This proof strategy will not usually succeed in proving consistency, because many equational theories cannot be completed at all, or cannot be completed in an acceptable amount of time and space. However, it has proved useful in debugging specifications, both for equational and non-equational traits.

In general, we can use LP's forward inference mechanisms to search for the presence of inconsistencies in a specification. The completion procedure can search for inconsistencies automatically, and we can instantiate axioms by "focus objects" (in the sense of McAllester [14]) to provide the completion procedure with a basis for its search. Even though unsuccessful searches do not certify that a specification is consistent, they increase our confidence in a specification, just as testing increases our confidence in a program.

Until recently, LSL allowed for another kind of claim that also involved a check for nonconsequence, namely a claim that one trait incorporated another without further constraining the meanings of its operators. However, none of our approaches to certifying or falsifying claims about "conservative extension" have proved practical. Without any promising means of checking them, we are inclined to consider claims about module independence as comments rather than as checkable claims, and have therefore removed from LSL the **imports** construct, which made such claims.

9 Extended example

To illustrate our approach to checking specifications in a slightly more realistic setting, we show how one might construct and check some traits to be used in the specification of a simple windowing system [8]. These are preliminary versions of traits that would likely be expanded as the specifications (including the interface parts) are developed.

The first three traits declare the signatures of some basic operators.

Coordinate: **trait**
  introduces
    $origin$: $\to$ $Coord$
    $\_\_ - \_\_$: $Coord, Coord \to Coord$
  asserts forall cd: $Coord$
    $cd - cd == origin$

Region(R): **trait**
  assumes *Coordinate*
  introduces
    $\_\_ \in \_\_$: $Coord, R \to Bool$
    % $cd \in r$ is true if point cd is in region r
    % Nothing is assumed about shape or contiguity of regions

Displayable(T): **trait**
  assumes *Coordinate*
  includes *Region*(T)
  introduces
    $\_\_[\_\_]$: $T, Coord \to Color$
    % $t[cd]$ represents the appearance of object t at point cd

The proof obligations associated with these traits are easily discharged. When LP's completion procedure is run on *Coordinate*, it terminates without generating any critical pairs. Since *Coordinate* has no **generated by** or **partitioned by** clauses, this is sufficient to guarantee that it is consistent. When checking the inclusion of *Region* by *Displayable*, *Region*'s assumption of *Coordinate* is discharged syntactically, using *Displayable*'s assumption of the same trait.

The next trait defines a window as an object composed of content and clipping regions, foreground and background colors, and a window identifier. $\in$ is qualified by a signature in the last line of the trait because it overloaded, and it is necessary to say which $\in$ is converted.

Window: **trait**
  assumes *Coordinate*
  includes *Region*, *Displayable*(W)
  asserts
    W **tuple of** *cont, clip*: *R*, *fore, back*: *Color*, *id*: *WId*
    forall w: W, cd: *Coord*
      $cd \in w == cd \in w.clip$
      $w[cd] == if\ cd \in w.cont\ then\ w.fore\ else\ w.back$
  implies converts $\_\_[\_\_]$, $\in$: $Coord, W \rightarrow Bool$

There are three proof obligations associated with this trait. The assumptions of *Coordinate* in *Region* and *Displayable* are syntactically discharged using *Window*'s assumption. The **converts** clause is discharged by LP without user assistance. The other proof obligation is consistency. As discussed in the previous section, we use the completion procedure to search for inconsistencies. Running it for a short time neither uncovers an inconsistency nor proves consistency.

The *View* trait (Figure 12) defines a view as an object composed of windows at locations. There are several proof obligations associated with this trait. Once again, the assumptions of *Window* and *Displayable* are discharged syntactically by the assumption in *View*. Once again, using the completion procedure to search for inconsistencies uncovers no problems. However, checking the **converts** clause turns up a problem. The conversion of *inW* and both $\in$'s is easily proved by induction over objects of sort *V*. However, the inductive base case for $\_\_[\_\_]$ does not reduce at all, because $emptyV[cd]$ is not defined. This problem can be solved either by defining $emptyV[cd]$ or by adding

  exempting forall cd: $Coord \quad emptyV[cd]$

to the **converts** clause. We choose the latter because there is no obvious definition for $emptyV[cd]$. With the added exemption, the inductive proof of the conversion of $\_\_[\_\_]$ goes through without further interaction.

When we attempt to prove the first of the explicit equations in the **implies** clause of *View*, we run into difficulty. After automatically applying its proof method for implications, LP reduces the conjecture to

  if $(cd'_0 - cd_0) \in w_0.clip$
    then if $(cd'_0 - cd_0) \in w_0.cont$
      then $w_0.fore$ **else** $w_0.back$
    else $v_0[cd'_0]$
  $== v_0[cd'_0]$

View: **trait**

assumes *Coordinate*

includes *Window*, *Displayable*(*V*)

introduces

$emptyV$: $\rightarrow\ V$

$addW$: $V, Coord, W \rightarrow V$

$\_\_ \in \_\_$: $W, V \rightarrow Bool$

inW: $V, WId, Coord \rightarrow Bool$

asserts

V **generated by** *emptyV, addW*

forall cd, cd': *Coord*, v: *V*, w, w': *W*, wid: *WId*

$\neg(cd \in emptyV)$

$cd \in addW(v, cd', w) == (cd - cd') \in w \mid cd \in v$

$\neg(w \in emptyV)$

$w \in addW(v, cd', w') == w.id = w'.id \mid w \in v$

$addW(v, cd', w)[cd] ==$

if $(cd - cd') \in w$ **then** $w[cd - cd']$ **else** $v[cd]$

% In view only if in a window, offset by origin

$\neg inW(emptyV, wid, cd)$

$inW(addW(v, cd, w), wid, cd') ==$

$(w.id = wid \ \&\ (cd - cd') \in w) \mid inW(v, wid, cd')$

implies

forall cd, cd': $Coord$, v, v': V, w: W

% Adding a new window does not affect the appearance

% of parts of the view lying outside the window

$\neg inW(addW(v, cd, w), w.id, cd') \Rightarrow$

$addW(v, cd, w)[cd'] = v[cd']$

% Appearance within a newly added window is independent

% of the view to which it is added.

$inW(addW(v, cd', w), w.id, cd) \Rightarrow$

$addW(v, cd', w)[cd] = addW(v', cd', w)[cd]$

converts inW, $\in$: $Coord, V \rightarrow Bool$, $\in$: $W, V \rightarrow Bool$,

$\_\_[\_\_]$: $V, Coord \rightarrow Color$

Figure 12: Sample View Specification

and reduces the assumed hypothesis of the implication to

$$\neg((cd_0 - cd'_0) \in w_0.clip)$$

At this point, we ask ourselves why the hypothesis is not sufficient to reduce the conjecture to an identity. The problem seems to be the order of the operands of $-$. This leads us to look carefully at the second equation for *inW* in trait *View*. There we discover that we have written $cd - cd'$ when we should have written $cd' - cd$, or, to be consistent with the form of the other equations, reversed the role of cd and cd' in the left hand side of the equation. After changing this axiom to

$$inW(addW(v, cd', w), wid, cd) == (w.id = wid \& (cd - cd') \in w) \mid inW(v, wid, cd)$$

the proof of the first implication goes through without interaction.

The second conjecture, after LP applies its proof method for implications, reduces to

$$\begin{array}{l}
\textbf{if } (cd_0 - cd'_0) \in w_0.clip \\
\quad \textbf{then if } (cd_0 - cd'_0) \in w_0.cont \\
\quad\quad \textbf{then } w_0.fore \textbf{ else } w_0.back \\
\quad \textbf{else } v_0[cd_0] \\
== \\
\textbf{if } (cd_0 - cd'_0) \in w_0.clip \\
\quad \textbf{then if } (cd_0 - cd'_0) \in w_0.cont \\
\quad\quad \textbf{then } w_0.fore \textbf{ else } w_0.back \\
\quad \textbf{else } v'[cd_0]
\end{array}$$

We resume the proof by dividing it into two cases based on the Boolean expression in the outermost **if**'s. When this expression is true, the conjecture reduces to *true*; when it is false, the conjecture reduces to

$$v_0[cd_0] == v'[cd_0]$$

Since v' is a variable and v_0 a fresh constant, we know that we are not going to be able to reduce this to *true*. This does not necessarily mean that the proof will fail, since we could be in an impossible case (that is to say, the current hypotheses could lead to a contradiction). However, examining the current hypotheses,

$$\begin{array}{ll}
inW(v_0, w_0.id, cd'_0) & \text{\% Implication hypothesis} \\
\neg((cd_0 - cd'_0) \in w_0.clip) & \text{\% Case hypothesis}
\end{array}$$

gives us no obvious reason to believe that a contradiction exists.

This leads us to wonder about the validity of the conjecture we are trying to prove, and to ask ourselves why we thought it was true when we added it to the trait. Our informal reasoning had been:

1. The hypothesis of the conjecture, $inW(addW(v, cd', w), wid, cd)$, guarantees that coordinate cd is in window w in the view $addW(v, cd', w)$.
2. If w is added at the same place in v' as in v, cd must also be in $addW(v', cd', w)$.
3. Furthermore $cd - cd'$ will be the same relative coordinate in w in both $addW(v, cd', w)$ and $addW(v', cd', w)$.
4. Therefore the equation

   $$\begin{array}{l} inW(addW(v, cd, w), wid, cd') == \\ \quad (w.id = wid \ \& \ (cd - cd') \in w) \mid inW(v, wid, cd') \end{array}$$

   in trait *View* should guarantee the conclusion.

The first step in formalizing this informal argument is to attempt to prove

$$inW(addW(v, cd', w), w.id, cd) \Rightarrow (cd - cd') \in w$$

as a lemma. LP reduces this conclusion of this implication to

$$(cd_0 - cd'_0) \in w_0.clip$$

using the normalized implication hypothesis

$$(cd_0 - cd'_0) \in w_0.clip \mid inW(v_0, w_0.id, cd'_0)$$

Casing on the first disjunct of the hypothesis reduces the conjecture to *false* under the same implication and case hypotheses as above.

We are thus stuck in the same place as in our attempted proof of the original conjecture. This leads us to question the validity of the first step in our informal proof, and we discover a flaw there: when v contains a window with the same id as w, the implication is not sound. The problem is that we implicitly assumed the invariant that no view would contain two windows with the same id, and our specification does not guarantee this. To correct this problem, we try adding an additional operator

$$numW\colon V, WId \to Nat$$

and three additional axioms

$$\begin{array}{ll} numW(emptyV, wid) == 0 & \\ numW(addW(v, cd', w), wid) == & \\ \quad numW(v, wid) + (if\ w.id = wid\ then\ 1\ else\ 0) & \\ numW(v, wid) \leq 1 & \text{\% New invariant} \end{array}$$

to the trait *View*. Unfortunately, when we run LP's completion procedure on the revised specification of *View*, we quickly get an inconsistency. There are several ways around this problem, among them:

1. Trait *View* could be changed so that *addW* chooses a unique *id* whenever a window is added.
2. Trait *View* could be changed so that *addW* is the identity function when the *id* of the window to be added is already associated with a window in the view.
3. The preservation of the invariant could be left to the interface level.

CartesianView: **trait**
  includes *View*, *NaturalNumber*
  asserts
    Coord **tuple of** x: Nat, y: Nat
    forall cd, cd': *Coord*
      $origin == [0, 0]$
      $cd - cd' == [cd.x - cd'.x,\ cd.y - cd'.y]$
  implies converts $origin, -$

Figure 13: Sample Cartesian View Specification

The third alternative is consistent with the interface specification given in Section 2, and is the one chosen here. This causes us to weaken the second implication of trait $View$ to:

forall cd, cd': $Coord$, v, v': V, w: W
  % Appearance within a newly added window is independent
  % of the view to which it is added, provided that the window
  % is not already present in the view.
  $(\neg(w \in v)\ \&\ \neg(w \in v')\ \&\ inW(addW(v, cd', w), w.id, cd))$
    $\Rightarrow addW(v, cd', w)[cd] = addW(v', cd', w)[cd]$

which is proved with a small amount of user interaction.

Finally, in Figure 13, we introduce a coordinate system. LP uses the facts of the trait *NaturalNumber* (not shown) to automatically discharge the assumption of *Coordinate* that has been carried from level to level. LP requires no assistance to complete the proof that the coordinate operators are indeed converted.

Of course, for expository purposes, we have used an artificially simplified example. We also deliberately seeded some errors for LP to find. However, most of the errors discussed above occurred unintentionally as we wrote the example, and we did not notice them until we actually attempted the mechanical proofs. With larger specifications, we expect bugs to be more frequent and harder to find.

10 Conclusions

The Larch Shared Language includes several facilities for introducing checkable redundancy into specifications. These facilities were chosen to expose common classes of errors. They give specifiers a better chance of receiving diagnostics about specifications with unintended meanings, in much the same way that type systems give programmers a better chance of receiving diagnostics about erroneous programs.

A primary goal of Larch is to provide useful feedback to specifiers when there is something wrong with a specification. Hence we have designed LP primarily as a debugging tool. We are not overly troubled that detecting inconsistencies is generally quicker and easier than certifying consistency.

We expect to discover flaws in specifications by having attempted proofs fail. LP does not automatically apply backwards inference techniques, and it requires more user guidance than some other provers. Much of this guidance is highly predictable, e.g, proving the hypotheses of deduction rules as lemmas. Although it is tempting to supply LP with heuristics that would generate such lemmas automatically, we feel that it is better to leave the choice to the user. At many points in a proof, many heuristics will apply. In our experience, choosing the next step in a proof (for example, a case split or proof by induction), or deciding that the proof attempt should be abandoned, often depends upon knowledge of the application. LP cannot reasonably be expected to possess this knowledge, especially when we are searching for a counterexample to a conjecture, rather than attempting to prove it. In some cases, LSLC may be able to use its knowledge of the structure of specifications to generate some of the guidance (for example, using induction to prove a **converts** clause) that users must currently provide to LP.

The checkable redundancy that LSL encourages in specifications also supports regression testing as specifications evolve. When we change part of a specification (for example, to strengthen the assertions of one trait), it is important to ensure that the change does not have unintended side-effects. LP's facilities for detecting inconsistencies help us discover totally erroneous changes. Claims about other traits in the specification, which imply or assume the changed trait, can help us discover more subtle problems. If some of these claims have already been checked, LP's facilities for replaying proof scripts make it easy to recheck their proofs after a change—an important activity, but one that is likely to be neglected without mechanical assistance.

We are encouraged by our early experience in using LP to check LSL specifications, but it is clear that more work must be done on both LSLC and LP before non-experts can use them cost-effectively. We plan to investigate having LSLC discharge more proof obligations textually and to provide more guidance for LP in discharging the others. We plan to enhance LP to include many of the syntactic amenities present in LSL and to provide further facilities for proof management. More fundamentally, we plan to enhance the logic of LP, enabling it to reason about formulas with embedded quantifiers. Finally, we plan to continue improving the performance of LP, while reducing the amount of guidance it requires, particularly when we use it to reason about theories that include standard subtheories, such as the Booleans or the natural numbers.

Acknowledgments

We wish to thank Andres Modet and Martín Abadi for their help in understanding the semantics of Larch and in designing several semantic checks. Andres Modet was responsible for the first implementation of LSLC, and James Rauen and Alexander Esterkin for the second. James

Saxe and Jørgen Staunstrup provided helpful comments and suggestions based on their use of LP.

References

[1] Boyer, R. S. and Moore, J S. *A Computational Logic*, New York, Academic Press, 1979.

[2] Boyer, R. S. and Moore, J S. *A Computational Logic Handbook*, New York, Academic Press, 1988.

[3] Burstall, R. M. and Goguen, J. A. "Putting Theories Together to Make Specifications," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, 1045–1058.

[4] Garland, S. J. and Guttag, J. V. "Inductive Methods for Reasoning about Abstract Data Types," *Proceedings of the 15th ACM Conference on Principles of Programming Languages*, 1988, 219–228.

[5] Garland, S. J. and Guttag, J. V. "An Overview of LP, The Larch Prover," *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Chapel Hill, N.C., *Lecture Notes in Computer Science* **355**, Springer-Verlag, 1989, 137–151.

[6] Garland, S. J., Guttag, J. V. and Staunstrup, J. "Verification of VLSI Circuits Using LP," *Proceedings of the IFIP WG 10.2 Conference on the Fusion of Hardware Design and Verification*, North Holland, 1988.

[7] Garland, S. J. and Guttag, J. V. "Using LP to Debug Specifications," *Proceedings of the IFIP TC2/WG2.2/WG2.3 Working Conference on Programming Concepts and Methods*, Elsevier, 1990.

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

[9] Guttag, J. V. and Horning, J. J. "Report on the Larch Shared Language," *Science of Computer Programming* 6:2 (Mar. 1986), 103–134.

[10] Guttag, J. V. and Horning, J. J. "A Larch Shared Language Handbook," *Science of Computer Programming* 6:2 (Mar. 1986), 135–157 (revision to appear).

[11] Guttag, J. V., Horning, J. J., and Modet, A. "Revised Report on the Larch Shared Language," Digital Equipment Corporation Systems Research Center Report 58, May 1990.

[12] Guttag, J. V., Horning, J. J. and Wing, J. M. "An Overview of the Larch Family of Specification Languages," *IEEE Software* 2:5 (Sept. 1985), 24–36.

[13] Knuth, D. E. and Bendix, P. B. "Simple Word Problems in Universal Algebras," in *Computational Problems in Abstract Algebra*, J. Leech (ed.), Pergamon Press, Oxford, 1969, 263–297.

[14] McAllester, D. A. "Ontic: A Knowledge Representation System for Mathematics", MIT Artificial Intelligence Laboratory Technical Report 979.

[15] Paulson, L. C. *Logic and Computation: Interactive Proof with Cambridge LCF,* Cambridge University Press, Cambridge, 1987.

[16] Peterson, G. L. and Stickel, M. E. "Complete Sets of Reductions for Some Equational Theories," *JACM* 28:2 (Apr. 1981), 233–264.

[17] Sannella, D. and Tarlecki, A. "On Observational Equivalence and Algebraic Specification," *Mathematical Foundations of Software Development*, *Lecture Notes in Computer Science* 186 (1985), 308–322.

[18] Shoenfield, J. R. *Mathematical Logic*, Addison-Wesley Publishing Company, 1967.

[19] Staunstrup, J., Garland, S. J., and Guttag, J. V. "Compositional verification of VLSI circuits," *Proceedings of an International Workshop on Automatic Verification of Finite State Systems*, Grenoble, France, June 1989, *Lecture Notes in Computer Science*, Springer-Verlag.