Massachusetts
Institute of
Technology

July 1988-
June 1989

Laboratory for Computer Science Progress Report

26

Systematic Program Development

Academic Staff

J.V. Guttag, Group Leader

Research Staff

S.J. Garland

Graduate Students

D. N. Jackson	M. T. Vandevoorde
M. Reinhold	K. A. Yelick
W. Shang	

Undergraduate Students

J. Chen	J. Rauen
M. Lillibridge	

Support Staff

A. Rubin

Visitors

T. Nipkow M.C. Godfrey

12.1 Introduction

The Systematic Program Development, though small, has a diverse set of interests. These include programming methodology, programming multiprocessors, specification languages (in conjunction with researchers at the Digital Equipment Corporation), circuit verification (in conjunction with researchers at the Technical University of Denmark), automatic theorem proving, and high performance garbage collection.

12.2 The Larch Family of Specification Languages

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*).

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 12.1 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

---

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 12.1: Sample Larch/CLU Interface Specification

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 12.1, 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.
- 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.

12.3 The LP Theorem Proving System

LP has changed dramatically in the last year. We take this opportunity to present a fairly detailed overview of its current capabilities.

The basis for proofs in LP is a logical system consisting of equations, rewrite rules, operator theories, induction rules, and deduction rules, all expressed in a multisorted fragment of first-order logic. A logical system in LP is closely related to an LSL theory, but is 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.

12.3.1 Declarations

Sorts, operators, and variables play exactly the same roles in LP as they do in LSL. They must be declared, and operators can be overloaded. The syntax for operators at the moment is not as rich as in LSL, but we plan to rectify that. Unlike LSL, LP at present provides no scoping for variables.

12.3.2 Equations and Rewrite Rules

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, i.e., that 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 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

$$\begin{aligned}&(x * y) * z \rightarrow x * (y * z)\\&x * i(x) \rightarrow e\\&e * x \rightarrow x.\end{aligned}$$

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 (i.e., the propositions that can be proved by reduction to normal form) is always a subset of its equational theory (i.e., the propositions that follow logically from its equations and from its rewrite rules considered as equations). The proof mechanisms discussed below 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 builtin 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.

12.3.3 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, e.g., 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.

12.3.4 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, e.g., by the LP commands

declare sorts E, S
declare operators
  $\{\}\colon \rightarrow S$
  $\{\_\_\}\colon E \rightarrow S$
  $\_\_ \cup \_\_\colon S, S \rightarrow S$
  $insert\colon S, E \rightarrow S$
  ..
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; e.g.,

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

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.

12.3.5 Deduction Rules

LP subsumes the logical power of the **partitioned by** construct of LSL by allowing users to assert deduction rules, which LP uses to deduce equations from other equations and rewrite rules. 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 builtin &-splitting law

declare variables p, q: *Bool*
when $p \,\&\, q == true$ **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.

12.3.6 Proof Mechanisms in LP

This section provides a brief overview of the proof mechanisms in LP.

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, e.g., to immune equations.
- The computation of *critical pairs* and the Knuth-Bendix *completion* procedure 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. However, we often make selective use of critical pairs. 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 $a < (b + c) \rightarrow \mathit{true}$ and $(b + c) < d \rightarrow \mathit{true}$, instantiating the deduction rule

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

  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 $a < d \rightarrow \mathit{true}$.

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, i.e., lemmas to be proved 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 above.

- *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 & ... & 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.

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 (e.g., 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, it does not perform heuristic searches for a proof. Unlike LCF, 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."

12.4 Hardware Verification

For many years, engineers have used simulation to convince themselves that the circuits they design behave as intended. As circuits get more complex, it becomes tempting to augment simulation with formal proofs. Typically, these proofs involve a large number of simple steps. Doing them by hand is cumbersome, boring, and prone to mistakes. Unless these proofs are machine generated or machine checked, there is very little reason to believe them.

In the past year, we have conducted several successful experiments using a theorem prover, LP, to verify properties of VLSI circuits. We started with several circuits that had previously been verified by hand. We then tried to construct machine checked proofs with the same structure as the original proofs. In the process of using LP to verify the circuits, we uncovered several minor errors in, and simplifications to, the original circuits and manual proofs.

Any formalized verification of a circuit must be based on an abstract description of the circuit. The choice of descriptive mechanism depends upon the intended use. Differential equations, for example, are useful in verifying physical properties such as power consumption, timing, or heat dissipation. Our approach is aimed at verifying functional properties of a design, and is based on describing the circuit as a parallel program, using a language, Synchronized Transitions, developed by Jørgen Staunstrup of the Technical University of Denmark.

While the circuits we have verified are not particularly complex, our experiments yielded several interesting insights. These include:

- Even for simple circuits, one cannot rely on proofs that have not been machine checked.
- Combined with Synchronized Transitions, the technique of invariant assertions used to verify safety properties of concurrent programs is useful for machine-aided reasoning about circuits.
- The verification process is quite sensitive to the exact way in which the problem is formulated. For example, proofs seem to work better when induction can be done over the structure of the circuit rather than over time.
- Circuit verification seems more amenable to machine checking than traditional program verification.
- The style of mechanical theorem proving supported by LP seems well suited to reasoning about circuits.

12.5 Programming Multiprocessors

In this research, we consider the problem of writing explicitly parallel programs for small to medium sized multiprocessors. To limit the scope of the work, the target applications are assumed to be symbolic problems, which are characterized by data structures that are irregular and dynamic and by a low percentage of numerical operations. In addition, we consider only applications for which a sequential solution is possible, i.e., concurrency is used to improve performance but is not inherent in the problem definition.

Various methods have been proposed to address the problem of writing and reasoning about parallel programs, but these tend to address only single module programs. Notions of module composition are missing and as a result, a style of program development is encouraged in which the entire program is designed and implemented as a single unit. Conversely, methods that have been developed for "programming in the large" of sequential programs are not applicable, and attempts to extend them to parallel programs often result in programs that exhibit very little real concurrency. Our goal is to be able to decompose parallel programs into independently specifiable units without prohibiting efficient implementations.

The research is organized around an extended example application, which involves parallel algorithm synthesis, correctness arguments, program module specifications, an implementation, and performance measurements. The application is to solve the completion problem for term rewriting systems, for which the well known Knuth and Bendix procedure is a sequential solution. Although the completion problem has been studied extensively, there are currently no parallel solutions.

Our parallel solution can be abstractly described by a set of inference rules that are nondeterministically applied to a system of rewrite rules and equations. In recent years, there has been a trend toward describing sequential completion procedures in this manner, often leading to better algorithms and simpler correctness arguments. The framework is especially useful in the context of concurrency, since there are known techniques for reasoning about concurrent systems using the possible sequences of state transitions. In this respect, our set of inference rules defines the set of possible state transitions, but there is an important difference in how we intend to reason about these systems. Rather than reasoning about the behavior of each high level transition by considering directly the sequence of low level operations that implement it, we intend to use the specifications of the underlying objects and their operations; each object is thus an independent unit which can be reused in any other context for which the same specification is required.

The implementation effort is still in progress but has already uncovered a number of interesting tradeoffs in the general programming problem. For example, concurrent data types that present the illusion of sequential access often lead to poor performance. The performance may be characterized by either long latency of operations, or little actual concurrency of multiple operations. In addition, examples of highly concurrent data types tend to have complex, almost mysterious implementations. In some cases, specification detail can be added to allow implementations that are more efficient or less complex, but for this we pay a price in terms of generality of the abstraction. We currently have an implementation of a completion procedure that exploits a small amount of parallelism, and closely mimics the behavior of a sequential implementation. As we add more parallelism to the procedure, the mapping between the parallel and sequential solutions will become less obvious, and the correctness argument more difficult.

12.6 Logic Programming

Most of the theoretical work on the semantics of logic programs assumes an interpreter that provides a complete resolution procedure. In contrast, for reasons of efficiency, most logic programming languages are built around incomplete procedures. This difference is rooted in Prolog, which evaluates resolvent trees in a depth-first rather than a breadth-first order. The gap is widened by some equational logic languages, which combine the incompleteness of depth-first evaluation with incomplete approximations to equational unification. Because of this gap, it is unsound to reason about logic programs using their declarative semantics. This in turn makes it difficult to develop abstraction mechanisms that can be used to partition a logic program into independently specifiable modules.

In this work, we considered the role type systems can play in closing the gap between the operational and declarative semantics of logic programs. We develop the notion of an equational mode system for use in constraining the domains of both predicates and unification procedures. The mode system is used to guide the resolution-based interpreter, and as a result, we can show that two predicate implementations with the same declarative meaning will be operationally equivalent.

This work was done in conjunction with Joseph Zachary of the University of Utah.

12.7 Publications

[1] S.J. Garland, J.V. Guttag, and J.S. Staunstrup. Verification of VLSI circuits using LP. In *Proceedings of International Workshop on Design for Behavioral Verification*, Glasgow, Scotland, July 1988.

[2] S.J. Garland and J.V. Guttag. An overview of LP, the Larch prover. In *Third International Conference on Rewriting Techniques and Applications*, Chapel Hill, NC, May 1989.

[3] S.J. Garland, J.V. Guttag, and J.S. Staunstrup. Compositional verification of VLSI circuits. In *Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.

[4] M.B. Reinhold. *Type Checking is Undecidable When "Type" is a Type*, Technical Report MIT/LCS/TR-458, MIT Laboratory for Computer Science, December 1989.

[5] M.T. Vandevoorde and E.S. Roberts. WorkCrews: an abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4), August 1988.

[6] K.A. Yelick and J.L. Zachary. Moded type systems for logic programming. In *Proceedings of Principles of Programming Languages Conference*, January 1989.

[7] P. Zave and D. Jackson. Practical specification techniques for control-oriented systems. In *Proceedings of the IFIP 11th World Computer Congress*, San Francisco, CA, 1989.