Simplify: A Theorem Prover for Program Checking

DAVID DETLEFS, GREG NELSON, AND JAMES B. SAXE

Hewlett-Packard

Abstract. This article provides a detailed description of the automatic theorem prover Simplify, which is the proof engine of the Extended Static Checkers ESC/Java and ESC/Modula-3. Simplify uses the Nelson–Oppen method to combine decision procedures for several important theories, and also employs a matcher to reason about quantifiers. Instead of conventional matching in a term DAG, Simplify matches up to equivalence in an E-graph, which detects many relevant pattern instances that would be missed by the conventional approach. The article describes two techniques, error context reporting and error localization, for helping the user to determine the reason that a false conjecture is false. The article includes detailed performance figures on conjectures derived from realistic program-checking problems.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Theorem proving, decision procedures, program checking

1. *Introduction*

This is a description of Simplify, the theorem prover used in the Extended Static Checking project (ESC) [Detlefs et al. 1998; Flanagan et al. 2002]. The goal of ESC is to prove, at compile-time, the absence of certain run-time errors, such as out-of-bounds array accesses, unhandled exceptions, and incorrect use of locks. We and our colleagues have built two extended static checkers, ESC/Modula-3 and ESC/Java, both of which rely on Simplify. Our ESC tools first process source code with a *verification condition generator*, producing first-order formulas asserting the absence of the targeted errors, and then submit those *verification conditions* to Simplify. Although designed for ESC, Simplify is interesting in its own right and has been used for purposes other than ESC. Several examples are listed in the conclusions.

Authors' addresses: D. Detlefs, Mailstop UBUR02-311, Sun Microsystems Laboratories, One Network Drive, Burlington, MA 01803-0902, e-mail: david.detlefs@sun.com; G. Nelson (Mailstop 1203) and J. B. Saxe (Mailstop 1250), Hewlett-Packard Labs, 1501 Page Mill Rd., Palo Alto, CA 94304, e-mail: {gnelson,jim.saxe}@hp.com.

Simplify's input is an arbitrary first-order formula, including quantifiers. Simplify handles propositional connectives by backtracking search and includes complete decision procedures for the theory of equality and for linear rational arithmetic, together with some heuristics for linear integer arithmetic that are not complete but have been satisfactory in our application. Simplify's handling of quantifiers by pattern-driven instantiation is also incomplete but has also been satisfactory in our application. The semantics of John McCarthy's functions for updating and accessing arrays [McCarthy 1963, Sec. 9] are also predefined. Failed proofs lead to useful error messages, including counterexamples.

Our goal is to describe Simplify in sufficient detail so that a reader who reimplemented it from our description alone would produce a prover which, though not equivalent to Simplify in every detail, would perform very much like Simplify on program-checking tasks. We leave out a few of the things that "just grew", but include careful descriptions of all the essential algorithms and interfaces. Readers who are interested in more detail than this article provides are free to consult the source code on the Web [Detlefs et al. 2003b].

In the remainder of the introduction, we provide an overview of the Simplify approach and an outline of the rest of the report.

When asked to check the validity of a conjecture G, Simplify, like many theorem provers, proceeds by testing the satisfiability of the negated conjecture $\neg G$.

To test whether a formula is satisfiable, Simplify performs a backtracking search, guided by the propositional structure of the formula, attempting to find a *satisfying assignment*—an assignment of truth values to atomic formulas that makes the formula true and that is itself consistent with the semantics of the underlying theories. Simplify relies on domain-specific algorithms for checking the consistency of the satisfying assignment. These algorithms will be described later; for now, we ask the reader to take for granted the ability to test the consistency of a satisfying assignment.

For example, to prove the validity of the conjecture G:

$$x < y \;\Rightarrow\; (x - 1 < y \wedge x < y + 2),$$

we form its negation, which, for purposes of exposition, we will rewrite as

$$x < y \wedge (x - 1 \geq y \vee x \geq y + 2).$$

The literals that appear in $\neg G$ are

$$x < y$$
$$x - 1 \geq y$$
$$x \geq y + 2.$$

Any assignment of truth values that satisfies $\neg G$ must have $x < y$ true, so the backtracking search begins by postulating $x < y$. Then, the search must explore two possibilities, either $x - 1 \geq y$ or $x \geq y + 2$ must be true. So the search proceeds as follows:

```
assume x < y.
case split on the clause x − 1 ≥ y ∨ x ≥ y + 2
    first case, assume x − 1 ≥ y
        discover the inconsistency of x < y ∧ x − 1 ≥ y
        backtrack from the first case (discard the assumption x − 1 ≥ y)
```

second case, assume $x \geq y + 2$
  discover the inconsistency of $x < y \land x \geq y + 2$
  backtrack from the second case
(having exhausted all cases and finding no satisfying assignment, . . .)
report that $\neg G$ is unsatisfiable, hence G is valid

In summary, the basic idea of the backtracking search is that the set of paths to explore is guided by the propositional structure of the conjecture; the test for consistency of each path is by domain-specific algorithms that reflect the semantics of the operations and predicates of particular theories, such as arithmetic. Simplify handles quantified formulas with a *matcher* that heuristically chooses relevant instances.

Section 2 describes Simplify's built-in theory and introduces notation and terminology. Section 3 describes the backtracking search and the heuristics that focus it. Section 4 gives a high-level description of the domain-specific decision procedures. Section 5 describes the additional machinery for handling quantified formulas, including some modifications to the search heuristics described in Section 3. Section 6 describes the methods used by Simplify to report the reasons that a proof has failed, an important issue that is often neglected. Sections 7 and 8 give details of the two most important domain-specific decision procedures, the E-graph and Simplex modules. Section 9 presents various measurements of Simplify's performance. Sections 10 through 12 discuss related and future work, summarize our experiences, and offer conclusions.

To help the reader find definitions of technical terms, we annotate some (not all) uses of technical terms with terse cross-references of the form (§1) to the section or subsection containing the term's definition. There is also a short index of selected identifiers in Appendix A.

2. *Simplify's Built-in Theory*

This section has two purposes. The first is to define Simplify's underlying theory more precisely. The second is to introduce some terminology that will be useful in the rest of the article.

The input to Simplify is a formula of untyped first-order logic with function and relations, including equality. That is, the language includes the *propositional connectives* $\land$, $\lor$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$; the *universal quantifier* $\forall$, and the *existential quantifier* $\exists$. Simplify requires that its input be presented as a symbolic expression as in Lisp, but, in this article, we will usually use more conventional mathematical notation.

Certain function and relation symbols have predefined semantics. It is convenient to divide these function and relation symbols into several theories.

First is the theory of equality, which defines the semantics of the equality relation $=$. Equality is postulated to be a reflexive, transitive, and symmetric relation that satisfies Leibniz's rule: $x = y \Rightarrow f(x) = f(y)$, for any function f.

Second is the theory of arithmetic, which defines the function symbols $+$, $-$, $\times$ and the relation symbols $>$, $<$, $\geq$, and $\leq$. These function symbols have the usual meaning; we will not describe explicit axioms. Simplify makes the rule that any terms that occur as arguments to the functions or relations of the arithmetic theory are assumed to denote integers, so that, for example, the following formula is

considered valid:

$$(\forall x : x < 6 \Rightarrow x \leq 5).$$

Third is the theory of maps, which contains the two functions *select* and *store* and the two axioms:

$$(\forall\, a, i, x : select(store(a, i, x), i) = x)$$
$$(\forall a, i, j, x : i \neq j \Rightarrow select(store(a, i, x), j) = select(a, j)).$$

These are called the *unit* and *non-unit select-of-store* axioms respectively. In our applications to program checking, maps are used to represent arrays, sets, and object fields, for example. We write $f[x]$ as shorthand for $select(f, x)$.

Fourth, because reasoning about partial orders is important in program-checking applications, Simplify has a feature to support this reasoning. Because Simplify's theory of partial orders is somewhat different from its other theories, we postpone its description to Section 4.7.

Simplify's theory is untyped, so that expressions that are intuitively mistyped, like $select(6, 2)$ or $store(a, i, x) + 3$ are legal, but nothing nontrivial about such terms is deducible in Simplify's built-in theory.

While our theory is untyped, we do draw a distinction between *propositional values* and *individual values*. The space of propositional values has two members, denoted by the *propositional literal constants* **true** and **false**. The space of individual values includes integers and maps. The *individual literal constants* of our language are the *integer literals* (like 14 and −36) and a special constant **@true** that we sometimes use to reflect the propositional value **true** into the space of individual values.

Since there are two kinds of values, there are two kinds of variables, namely *propositional variables*, which range over {**true**, **false**}, and *individual variables*, which range over the space of individual values. All bound variables introduced by quantifiers are individual variables.

A *term* is an individual variable, an individual literal constant, or an application of a function symbol to a list of terms. A term is a *ground term* if it contains no quantified variables.

An *atomic formula* is a propositional variable, a propositional literal constant, or the application of a relation to a list of terms. In Sections 3.3 and 5.3.1, we will define special propositional variables called *proxies*. Although a proxy is not represented as an identifier, in the syntactic taxonomy that we are currently describing, a proxy is an atomic formula, like any other propositional variable.

The strict distinction between terms and formulas (and thus also between functions and relations and between individual and propositional variables) is a feature of the classical treatment of first-order logic, so maintaining the distinction seemed a safe design decision in the early stages of the Simplify project. In fact, the strict distinction became inconvenient on more than one occasion, and we are not sure if we would make the same decision if we could do it over again. But we aren't sure of the detailed semantics of any alternative, either.

When the strict distinction between functions and relations enforced by Simplify is awkward, we circumvent the rules by modelling a relation as a function whose result is equal to **@true** iff the relation holds of its arguments. We call such a function a *quasi-relation*. For example, to say that the unary quasi-relation f is the pointwise conjunction of the unary quasi-relations g and h,

we could write

$$(\forall x : f(x) = \textbf{@true} \iff g(x) = \textbf{@true} \wedge h(x) = \textbf{@true}).$$

As a convenience, Simplify accepts the command (DEFPRED (f x)), after which occurrences of $f(t)$ are automatically converted into $f(t) = \textbf{@true}$ if they occur where a formula is expected.

The DEFPRED facility has another, more general, form:

$$(\text{DEFPRED } (r \ args) \ body), \tag{1}$$

which, in addition to declaring r to be a quasi-relation, also declares the meaning of that relation, the same meaning as would be declared by

$$(\forall args : r(args) = \textbf{@true} \iff body). \tag{2}$$

However, although (1) and (2) give the same meaning to r, there is a heuristic difference in the way Simplify uses them. The quantified form 2 is subjected to pattern-driven instantiation as described in Section 5.1. In contrast, the formula 1 is instantiated and used by Simplify only when an application of r is explicitly equated to **@true**. This is explained further in one of the fine points in Section 7.

An *equality* is an atomic formula of the form $t = u$ where t and u are terms. An *inequality* is an atomic formula of one of the forms

$$t \leq u, \quad t < u, \quad t \geq u, \quad t > u$$

where t and u are terms. A *binary distinction* is an atomic formula of the form $t \neq u$, where t and u are terms. A *general distinction* is an atomic formula of the form $\text{DISTINCT}(t_1, \ldots, t_n)$ where the t's are terms; it means that no two of the t's are equal.

The atomic formulas of Simplify's theory of equality are the equalities and the distinctions (binary and general). We allow general distinctions because (1) in our applications they are common, (2) expressing a general distinction in terms of binary distinctions would require a conjunction of length $O(n^2)$, and (3) we can implement general distinctions more efficiently by providing them as a primitive.

The atomic formulas of Simplify's theory of arithmetic are the inequalities and equalities.

Simplify's theory of maps is characterized by the postulated semantics of the function symbols *store* and *select*. It has no relation symbols of its own, and its atomic formulas are simply the atomic formulas of the theory of equality.

A *formula* is an expression built from atomic formulas, propositional connectives, and quantifiers.

The formula presented to Simplify to be proved or refuted is called the *conjecture*. The negation of the conjecture, which Simplify attempts to satisfy, is called the *query*.

Some particular kinds of formulas are of special importance in our exposition. A *literal* is an atomic formula or the negation of an atomic formula. This atomic formula is called the *atom* of the literal. A *clause* is a disjunction of literals, and a *monome* is a conjunction of literals. A *unit clause* is a clause containing a single literal.

A *satisfying assignment* for a formula is an assignment of values to its free variables and of functions to its free function symbols, such that the formula is

true if its free variables and function symbols are interpreted according to the assignment, and the built-in functions satisfy their built-in semantics.

A formula is *satisfiable* (or *consistent*) if it has a satisfying assignment, and *valid* if its negation is not satisfiable.

An important fact on which Simplify relies is that a formula such as

$$(\forall x : (\exists y : P(x, y)))$$

is satisfiable if and only if the formula

$$(\forall x : P(x, f(x))))$$

is satisfiable, where f is an otherwise unused function symbol. This fact allows Simplify to remove quantifiers that are essentially existential—that is, existential quantifiers in positive position and universal quantifiers in negative position—from the query, replacing all occurrences of the quantified variables with terms like $f(x)$ above. The function f is called a *Skolem* function, and this process of eliminating the existential quantifiers is called *Skolemization*. The arguments of the Skolem function are the essentially universally quantified variables in scope at the point of the quantifier being eliminated.

An important special case of Skolemization concerns free variables of the conjecture. All free variables of the conjecture are implicitly universally quantified at the outermost level, and thus are implicitly existentially quantified in the query. Simplify therefore replaces these variables with applications of nullary Skolem functions (also called *Skolem constants*).

In addition to providing decision procedures for a built-in theory used for all conjectures, Simplify allows users to supply an arbitrary formula as a *background predicate* that is given once and then used as an implicit antecedent for a number of different conjectures. Users can supply a background predicate containing axioms for a theory that is useful in their application. An example of the use of this facility is provided by ESC: many facts relating to the verification of a procedure are common to all the procedures in a given module; ESC assumes those facts in the background predicate and then checks the verification conditions (§1) of the procedures in the module one by one.

3. *The Search Strategy*

In this section, we describe Simplify's backtracking search strategy. Since the search strategy is essentially concerned with the propositional structure of the conjecture, we assume throughout this section that the conjecture is a propositional formula all of whose atomic formulas are propositional variables. Compared to recent advances in propositional SAT solving [Zhang 1997; Guerra e Silva et al. 1999; Silva and Sakallah 1999; Moskewicz et al. 2001], the backtracking search described in this section is simple and old-fashioned. We include this material not because it is a contribution by itself, but because it is the foundation for the later material in which domain-specific decision procedures and quantifiers are incorporated.

3.1. THE INTERFACE TO THE CONTEXT.   Simplify uses a global resettable data structure called the *context* which represents the conjunction of the query (§2) together with the assumptions defining the current case.

The context has several components, some of which will be described in later sections of the report. To begin with, we mention three of its components: the public

boolean *refuted*, which can be set any time the context is detected to be inconsistent; the *literal set*, *lits*, which is a set of literals; and the *clause set*, *cls*, which is a set of clauses, each of which is a set of literals. A clause represents the disjunction of its elements. The literal set, on the other hand, represents the conjunction of its elements. The entire context represents the conjunction of all the clauses in *cls* together with *lits*. When there is no danger of confusion, we shall feel free to identify parts of the context with the formulas that they represent. For example, when referring to the formula represented by the clause set, we may simply write "*cls*" rather than $\bigwedge_{c \in cls} (\bigvee_{l \in c} l)$.

The algorithm for determining satisfiability operates on the context through the following interface, called the *satisfiability interface*:

> **proc** *AssertLit*(*P* : *Literal*) ≡ add the literal *P* to *lits* and possibly
> set *refuted* if this makes *lits* inconsistent
> **proc** *Push*() ≡ save the state of the context
> **proc** *Pop*() ≡ restore the most recently saved, but
> not-yet-restored, context

AssertLit(*P*) "possibly" sets *refuted* when *lits* becomes inconsistent, because some of Simplify's decisions procedures are incomplete; it is always desirable to set *refuted* if it is sound to do so.

In addition, the context allows clauses to be deleted from the clause set and literals to be deleted from clauses.

Viewed abstractly, *Push* copies the current context onto the top of a stack, from which *Pop* can later restore it. As implemented, Simplify maintains an *undo stack* recording changes to the context in such a way that *Pop* can simply undo the changes made since the last unmatched call to *Push*.

At any point in Simplify's execution, those changes to the context that have been made since the beginning of execution but not between a call to *Push* and the matching call to *Pop* are said to have occurred "on the *current path*," and such changes are said to be *currently* in effect. For example, if a call *AssertLit*(*P*) has occurred on the current path, then the literal *P* is said to be *currently asserted*.

The data structures used to represent the context also store some heuristic information whose creation and modification is not undone by *Pop*, so that Simplify can use information acquired on one path in the backtracking search to improve its efficiency in exploring other paths. In the sections describing the relevant heuristics (scoring in Section 3.6 and promotion in Section 5.2.1) we will specifically note those situations in which changes are not undone by *Pop*.

Whenever the conjunction of currently asserted literals becomes inconsistent, the boolean *refuted* may be set to **true**. As we shall see in more detail later, this will cause the search for a satisfying assignment to backtrack and consider a different case. Were *refuted* to be erroneously set to **true**, Simplify would become unsound. Were it to be left **false** unnecessarily, Simplify would be incomplete. For now, we are assuming that all atomic formulas are propositional variables, so it is easy for *AssertLit* to ensure that *refuted* is **true** iff *lits* is inconsistent: a monome (§2) is inconsistent if and only if its conjuncts include some variable v together with its negation $\neg v$. The problem of maintaining *refuted* will become more challenging as we consider richer classes of literals.

3.2. THE *Sat* PROCEDURE. To test the validity of a conjecture G, Simplify initializes the context to represent $\neg G$ (as described in Sections 3.3 and 3.4), and then uses a recursive procedure *Sat* (described in this section) to test whether the context is satisfiable by searching exhaustively for an assignment of truth values to propositional variables that implies the truth of the context. If *Sat* finds such a satisfying assignment for the query $\neg G$, then Simplify reports it to the user as a counterexample for G. Conversely, if *Sat* completes its exhaustive search without finding any satisfying assignment for $\neg G$, then Simplify reports that it has proved the conjecture G.

The satisfying assignments found by *Sat* need not be total. It is often the case that an assignment of truth values to a proper subset of the propositional variables of a formula suffices to imply the truth of the entire formula regardless of the truth values of the remaining variables.

It will be convenient to present the pseudo-code for *Sat* so that it outputs a set of satisfying assignments covering all possible ways of satisfying the context, where each satisfying assignment is represented as a monome, namely the conjunction of all variables made **true** by the assignment together with the negations of all variables made **false**. Thus the specification of *Sat* is:

```
proc Sat()
/* Outputs zero or more monomes such that (1) each monome is consistent, (2) each
monome implies the context, and (3) the context implies the disjunction of all the monomes.
*/
```

Conditions (1) and (2) imply that each monome output by *Sat* is indeed a satisfying assignment for the context. Conditions (2) and (3) imply that the context is equivalent to the disjunction of the monomes, that is, *Sat*, as given here, computes a *disjunctive normal form* for the context. If Simplify is being used only to determine whether the conjecture is valid, then the search can be halted as soon as a single counterexample context has been found. In an application like ESC, it is usually better to find more than one counterexample if possible. Therefore, the number of counterexamples Simplify will search for is configurable, as explained in Section 6.3.

We implement *Sat* with a simple backtracking search that tries to form a consistent extension to *lits* by including one literal from each clause in *cls*. To reduce the combinatorial explosion, the following procedure (*Refine*) is called before each case split. The procedure relies on a global boolean that records whether refinement is "enabled"(has some chance of discovering something new). Initially, refinement is enabled.

```
proc Refine() ≡
    while refinement enabled do
    disable refinement;
    for each clause C in cls do
      RefineClause(C);
      if refuted then
        return
      end
    end
  end
end
```

```
proc RefineClause(C : Clause) ≡
    if C contains a literal l such that [lits ⇒ l] then
        delete C from cls;
        return
    end;
    while C contains a literal l such that [lits ⇒ ¬l] do
        delete l from C
    end;
    if C is empty then
        refuted := true
    else if C is a unit clause {l} then
        AssertLit(l);
        enable refinement
    end
end
```

We use the notation $[P \Rightarrow Q]$ to denote that Q is a logical consequence of P. With propositional atomic formulas, it is easy to test whether $[lits \Rightarrow l]$: this condition is equivalent to $l \in lits$. Later in this article, when we deal with Simplify's full collection of literals, the test will not be so easy. At that point (Section 4.6), it will be appropriate to consider imperfect tests. An imperfect test that yielded a false positive would produce an unsound refinement, and this we will not allow. The only adverse effect of a false negative, on the other hand, is to miss the heuristic value of a sound refinement, and this may be a net gain if the imperfect test is much more efficient than a perfect one.

Evidently *Refine* preserves the meaning of the context so that the sequence *Refine*(); *Sat*() meets the specification for *Sat*. Moreover, *Refine* has the heuristically desirable effects of

—removing clauses that are already satisfied by *lits* (called *clause elimination*),

—narrowing clauses by removing literals that are inconsistent with *lits* and thus inconsistent with the context (called *width reduction*), and

—moving the semantic content of unit clauses from *cls* to *lits*, (called *unit assertion*).

The pseudo-code above shows *Refine* employing its heuristics in a definite order, attempting first clause elimination, then width reduction, then unit propagation. In fact, the heuristics may be applied in any order, and the order in which they are actually applied by Simplify often differs from that given above.

Here is an implementation of *Sat*:

```
proc Sat() ≡
    enable refinement;
    Refine();
    if refuted then return
    else if cls is empty then
        output the satisfying assignment lits;
        return
    else
        let c be some clause in cls, and l be some literal of c;
        Push();
        AssertLit(l);
        delete c from cls;
        Sat()
```

```
        Pop();
        delete l from c;
        Sat()
    end
end
```

The proof of correctness of this procedure is straightforward. As noted above calling *Refine* preserves the meaning of the context. If *refuted* is **true** or *cls* contains an empty clause then the context is unsatisfiable and it is correct for *Sat* to return without emitting any output. If *cls* is empty, then the context is equivalent to the conjunction of the literals in *lits*, so it is correct to output this conjunction (which must be consistent, else the context would have been refuted) and return. If *cls* is not empty, then it is possible to choose a clause from *cls*, and if the context is not already refuted, then the chosen clause *c* is nonempty, so it is possible to choose a literal *l* of *c*. The two recursive calls to *Sat* are then made with contexts whose disjunction is equivalent to the original context, so if the monomes output by the recursive calls satisfy conditions (1)–(3) for those contexts, then the combined set of monomes satisfies conditions (1)–(3) for the original context.

Here are some heuristic comments about the procedure.

(1) The choice of which clause to split on can have an enormous effect on performance. The heuristics that govern this choice will be described in Sections 3.5, 3.6, and 5.2 below.

(2) The literal set is implemented (in part) by maintaining a *status* field for each atomic formula indicating whether that atomic formula's truth value is known to be **true**, known to be **false**, or unknown. The call *AssertLit(l)* normally sets the truth status of *l*'s atom (§2) according to *l*'s sense, but first checks whether it is already known with the opposite sense, in which case it records detection of a contradiction by setting the *refuted* bit in the context. The *refuted* bit is, of course, reset by *Pop*.

(3) A possible heuristic, which we refer to as the *subsumption heuristic*, is to call *Context.AssertLit(¬l)* before the second recursive call to *Sat* (since the first recursive call to *Sat* has exhaustively considered cases in which *l* holds, subsuming the need to consider any such cases in the second call).

The preceding pseudo-code is merely a first approximation to the actual algorithm employed by Simplify. In the remainder of this report, we will describe a number of modifications to *Sat*, to the procedures it calls (e.g., *Refine* and *AssertLit*), to the components of the context, and to the way the context is initialized before a top-level call to *Sat*. Some of these changes will be strict refinements in the technical sense—constraining choices that have so far been left nondeterministic; others will be more radical (e.g., weakening the specification of *Sat* to allow incompleteness when quantifiers are introduced).

In the remainder of this section, we describe the initialization of the context (at least for the case where the conjecture is purely propositional) and some heuristics for choosing case splits.

3.3. EQUISATISFIABLE CNF. We now turn to the problem of initializing the context to represent the query (§2), a process we sometimes refer to as *interning* the query.

The problem of initializing the context to be equivalent to some formula F is equivalent to the problem of putting F into conjunctive normal form (CNF), since the context is basically a conjunction of clauses.

It is well known that any propositional formula can be transformed into logically equivalent CNF by applying distributivity and DeMorgan's laws. Unfortunately, this may cause an exponential blow-up in size. Therefore, we do something cheaper: we transform the query Q into a formula that is in CNF, is linear in the size of Q and is equisatisfiable with Q. We say that two formulas F and G are *equisatisfiable* if "F is satisfiable" is equivalent to "G is satisfiable". In summary, we avoid the exponential blow-up by contenting ourselves with equisatisfiability instead of logical equivalence.

To do this, we introduce propositional variables, called *proxies*, corresponding to subformulas of the query, and write clauses that enforce the semantics of these proxy variables. For example, we can introduce a proxy X for $P \wedge R$ by introducing the clauses

$$\neg X \vee P$$
$$\neg X \vee R$$
$$X \vee \neg P \vee \neg R$$

whose conjunction is equivalent to

$$X \;\Leftrightarrow\; (P \wedge R).$$

We refer to the set of clauses enforcing the semantics of a proxy as the *definition* of the proxy. Note that these clauses uniquely determine the proxy in terms of the other variables.

Given a query Q, if we introduce proxies for all nonliteral subformulas of Q (including a proxy for Q itself) and initialize the context to contain the definitions of all the proxies together with a unit clause whose literal is the proxy for Q, then the resulting context will be satisfiable if and only if Q is. The history of this technique is traced to Skolem in the 1920s in an article by Bibel and Eder [1993].

During the interning process, Simplify detects repeated subformulas and represents each occurrence of a repeated subformula by the same proxy, which need be defined only once. Simplify makes a modest attempt at *canonicalizing* its input so that it can sometimes recognize subformulas that are logically equivalent even when they are textually different. For example, if the formulas

$$R \wedge P$$
$$P \wedge R$$
$$\neg P \vee \neg R$$

all occurred as subformulas of the query, their corresponding proxy literals would all refer to the same variable, with the third literal having the opposite sense from the first two. The canonicalization works from the bottom up, so if P and P' are canonicalized identically and R and R' are canonicalized identically, then, for example, $P \wedge R$ will canonicalize identically with $P' \wedge R'$. However, the canonicalization is not sufficiently sophisticated to detect, for example, that $(P \vee \neg R) \wedge R$ is equivalent to $P \wedge R$.

The *Sat* procedure requires exponential time in the worst case, even for purely propositional formulas. When combined with matching to handle formulas with quantification (as discussed in Section 5), it can fail to terminate. This is not

surprising, since the satisfiability problem is *NP*-complete even for formulas in propositional calculus, and the validity of arbitrary formulas in first-order predicate calculus is only semidecidable. These observations don't discourage us, since in the ESC application the inputs to Simplify are verification conditions (§1), and if a program is free of the kinds of errors targeted by ESC, there is almost always a short proof of the fact. (It is unlikely that a real program's lack of array bounds errors would be dependent on the four-color theorem; and checking such a program would be beyond our ambitions for ESC.) Typical ESC verification conditions are huge but shallow. Ideally, the number of cases considered by Simplify would be similar to the number of cases that would need to be considered to persuade a human critic that the code is correct.

Unfortunately, experience with our first version of Simplify showed that if we turned it loose on ESC verification conditions, it would find ways to waste inordinate amounts of time doing case splits fruitlessly. In the remainder of Section 3, we will describe techniques that we use to avoid the combinatorial explosion in practice.

3.4. Avoiding Exponential Mixing with Lazy CNF.   We discovered early in our project that Simplify would sometimes get swamped by a combinatorial explosion of case splits when proving a conjecture of the form $P \wedge Q$, even though it could quickly prove either conjunct individually. Investigation identified the problem that we call "exponential mixing": Simplify was mixing up the case analysis for P with the case analysis for Q, so that the total number of cases grew multiplicatively rather than additively.

We call our solution to exponential mixing "lazy CNF". The idea is that instead of initializing the clause set *cls* to include the defining clauses for all the proxies of the query, we add to *cls* the defining clauses for a proxy p only when p is asserted or denied. Thus, these clauses will be available for case splitting only on branches of the proof tree where they are relevant.

Lazy CNF provides a benefit that is similar to the benefit of the "justification frontier" of standard combinatorial search algorithms [Guerra e Silva et al. 1999]. Lazy CNF is also similar to the "depth-first search" variable selection rule of Barrett et al. [2002a].

Introducing lazy CNF into Simplify avoided such a host of performance problems that the subjective experience was that it converted a prover that didn't work into one that did. We did not implement any way of turning it off, so the performance section of this paper gives no measurements of its effect.

3.4.1. *Details of Lazy CNF.*   In more detail, the lazy CNF approach differs from the nonlazy approach in five ways.

First, we augment the context so that in addition to the clause set *cls* and the literal set *lits*, it includes a *definition set*, *defs*, containing the definitions for all the proxy variables introduced by equisatisfiable CNF and representing the conjunction of those definitions. Simplify maintains the invariant that the definition set uniquely specifies all proxy variables in terms of the nonproxy variables. That is, if $v_1, \ldots, v_n$ are the nonproxy variables and $p_1, \ldots, p_m$ are the proxy variables, *defs* will be such that the formula

$$\forall v_1, \ldots, v_n : \exists! \, p_1, \ldots, p_m : \textit{defs} \qquad (3)$$

is valid (where $\exists!$ means "exists uniquely"). The context as a whole represents

the formula

$$\forall p_1, \ldots, p_m : (\textit{defs} \Rightarrow (\textit{cls} \land \textit{lits}))$$

or equivalently (because of (3))

$$\exists p_1, \ldots, p_m : (\textit{defs} \land \textit{cls} \land \textit{lits}).$$

Second, we change the way that Simplify initializes the context before invoking *Sat*. Specifically, given a query Q, Simplify creates proxies for all non-literal sub-formulas of Q (including Q itself) and initializes the context so *lits* is empty, *cls* contains only a single unit clause whose literal is the proxy for Q, and *defs* contains definitions making all the proxy variables equivalent to the terms for which they are proxies (that is, *defs* is initialized so that

$$\forall v_1, \ldots, v_n, p_1, \ldots, p_m : \textit{defs} \Rightarrow (p_i \Leftrightarrow T_i)$$

is valid whenever p_i is a proxy variable for term T_i). It follows from the conditions given in this and the preceding paragraph that (the formula represented by) this initial context is logically equivalent to Q. It is with this context that Simplify makes its top-level call to *Sat*.

Third, we slightly modify the specification of *Sat*, as indicated by italics:

proc *Sat*()
/* Requires that all proxy literals in *lits* be redundant. Outputs zero or more monomes such that (1) each monome is a consistent conjunction of *non-proxy* literals, (2) each monome implies the context, and (3) the context implies the disjunction of the monomes. */

When we say that the proxy literals in *lits* are redundant, we mean that the meaning of the context would be unchanged if all proxy literals were removed from *lits*. More formally, if $plits$ is the conjunction of the proxy literals in *lits* and *nlits* is the conjunction of the nonproxy literals in *lits* then

$$(\forall p_1, \ldots, p_m : (\textit{defs} \land \textit{cls} \land \textit{nlits}) \Rightarrow \textit{plits}).$$

Fourth, we modify the implementation of *AssertLit* so that proxy literals are treated specially. When l is a nonproxy literal, the action of *AssertLit(l)* remains as described earlier: it simply adds l to *lits*, and possibly sets *refuted*. When l is a proxy literal, however, *AssertLit(l)* not only adds l to *lits*, but also adds the definition of l's atom (which is a proxy variable) to *cls*. As an optimization, clauses of the definition that contain l are not added to the clause set (since they would immediately be subject to clause elimination (§3.2)) and the remaining clauses are width-reduced by deletion of $\neg l$ before being added to the clause set. For example, suppose that s is a proxy for S, that t is a proxy for T, and p is a proxy for $S \land T$, so that the definition of p consists of the clauses:

$$\neg p \lor s$$
$$\neg p \lor t$$
$$p \lor \neg s \lor \neg t$$

Then, the call *AssertLit(p)* would add p to *lits* and add the unit clauses s and t to *cls*, while the call *AssertLit(¬ p)* would add the literal $\neg p$ to *lits* and add the clause $\neg s \lor \neg t$ to *cls*. It may seem redundant to add the proxy literal to *lits* in addition to adding the relevant width-reduced clauses of its definition to *cls*, and in fact, it would be unnecessary—if performance were not an issue. This policy of adding

the proxy literal to *lits* is called *redundant proxy assertion*. Its heuristic value will be illustrated by one of the examples later in this section.

Finally, we change *Sat* so that when it finds a satisfying assignment (§2) (i.e., when *cls* is empty and *lits* is consistent), the monome it outputs is the conjunction of only the non-proxy literals in *lits*. (Actually this change is appropriate as soon as we introduce proxy variables, regardless of whether or not we introduce their defining clauses lazily. Deleting the proxy literals from the monomes doesn't change the meanings of the monomes because the proxies are uniquely defined in terms of the nonproxy variables.)

To see how use of lazy CNF can prevent exponential mixing, consider proving a conjecture of the form $P_1 \wedge P_2$, where P_1 and P_2 are complicated subformulas. Let p_1 be a proxy for P_1, p_2 be a proxy for P_2, and p_3 be a proxy for the entire formula $P_1 \wedge P_2$.

In the old, nonlazy approach, the initial clause set would contain the unit clause $\neg p_3$; defining clauses for p_3, namely

$$\neg p_3 \vee p_1, \qquad \neg p_3 \vee p_2, \qquad p_3 \vee \neg p_1 \vee \neg p_2;$$

and the defining clauses for p_1, p_2, and all other proxies for subformulas of P_1 and P_2. The *Refine* procedure would apply unit assertion to assert the literal $\neg p_3$, and then would apply clause elimination to remove the first two defining clauses for p_3 and width reduction (§3.2) to reduce the third clause to

$$\neg p_1 \vee \neg p_2.$$

This clause and all the defining clauses for all the other proxies would then be candidates for case splitting, and it is plausible (and empirically likely) that exponential mixing would ensue.

In the new, lazy approach, the initial clause set contains only the unit clause $\neg p_3$. The *Refine* procedure performs unit assertion, removing this clause and calling *AssertLit*($\neg p_3$), which adds the clause

$$\neg p_1 \vee \neg p_2$$

to *cls*. At this point, *Refine* can do no more, and a case split is necessary. Suppose (without loss of generality) that the first case considered is $\neg p_1$. Then, *Sat* pushes the context, removes the binary clause, and calls *AssertLit*($\neg p_1$), adding $\neg p_1$ to the literal set and the relevant part of p_1's definition to the clause set. The refutation of $\neg P_1$ (recall that p_1 is logically equivalent to P_1, given *defs*) continues, perhaps requiring many case splits, but the whole search is carried out in a context in which no clauses derived from P_2 are available for case-splitting. When the refutation of $\neg P_1$ is complete, the case $\neg p_2$ (meaning $\neg P_2$) is considered, and this case is uncontaminated by any clauses from P_1. (Note that if Simplify used the subsumption heuristic (§3.2) to assert p_1 while analyzing the $\neg p_2$ case, the benefits of lazy CNF could be lost. We will say more about the interaction between lazy CNF and the subsumption heuristic in Section 3.7.)

As another example, let us trace the computation of Simplify on the conjecture

$$(s \wedge (s \implies t)) \implies t.$$

Simplify introduces proxies to represent the subformulas of the conjecture. Let us

call these p_1, p_2, p_3, p_4, and p_5, where

> p_1 is a proxy for $s \Rightarrow t$,
> p_2 is a proxy for $s \wedge (s \Rightarrow t)$, that is, for $s \wedge p_1$, and
> p_3 is a proxy for the entire conjecture, that is, for $p_2 \Rightarrow t$.

The context is initialized to the following state:

defs:
  definition of p_1:
    $p_1 \vee s$
    $p_1 \vee \neg t$
    $\neg p_1 \vee \neg s \vee t$
  definition of p_2:
    $\neg p_2 \vee s$
    $\neg p_2 \vee p_1$
    $p_2 \vee \neg s \vee \neg p_1$
  definition of p_3:
    $p_3 \vee p_2$
    $p_3 \vee \neg t$
    $\neg p_3 \vee \neg p_2 \vee t$

lits:

cls:
  $\neg p_3$

The *Refine* procedure first performs unit assertion on the clause $\neg p_3$, removing the clause $\neg p_3$ from the clause set, adding $\neg p_3$ to the literal set, and adding the unit clauses p_2 and $\neg t$ to the clause set. These clauses are subjected to unit assertion in turn: they are removed from the clause set, their literals are added to the literal set, and the unit clauses s and p_1 (from the definition of p_2) are added to the clause set. Applying unit assertion to these clauses leaves the context in the following state:

defs:
  (same as above)

lits:
  $\neg p_3$
  p_2
  $\neg t$
  s
  p_1

cls:
  $\neg s \vee t$

The only clause in the clause set is $\neg s \vee t$ (from the definition of p_1). Since both literals of this clause are negations of literals in the literal set, width reduction can be applied to reduce this clause to an empty clause, thus refuting the context. (Alternatively, the clause could be reduced to a unit clause—either to $\neg s$ or to t—after which unit assertion would make the literal set inconsistent, refuting the context). So we see that *Sat* can prove the conjecture

$$(s \wedge (s \Rightarrow t)) \Rightarrow t$$

entirely through the action of *Refine*, without the need for case splits.

Note that the proof would proceed in essentially the same way, requiring no case splits, for any conjecture of the form

$$(S_1 \wedge (S_2 \Rightarrow T_1)) \Rightarrow T_2$$

where S_1 and S_2 are arbitrarily complicated formulas that canonicalize (§3.3) to the same proxy literal s and where T_1 and T_2 are arbitrarily complicated formulas that canonicalize to the same proxy literal t. But this desirable fact depends on redundant proxy assertion, since without this policy, we would be left with the clause $\neg s \vee t$ in the clause set, and the literal proxies s and $\neg t$ would not have been introduced into the literal set. This illustrates the value of redundant proxy assertion.

3.5. GOAL PROXIES AND ASYMMETRICAL IMPLICATION.   Semantically, $P \Rightarrow Q$ is identical with $\neg P \vee Q$ and with $Q \vee \neg P$ and with $\neg Q \Rightarrow \neg P$. But heuristically, we have found it worthwhile to treat the two arms of an implication differently.

The reason for this is that the verification condition generator (§1), which prepares the input to Simplify, may sometimes have reason to expect that certain case splits are heuristically more desirable than others. By treating the consequent of an implication differently than the antecedent, we make it possible for the verification condition generator to convey this hint to Simplify.

For example, consider verifying a procedure whose postcondition is a conjunction and whose precondition is a disjunction:

```
proc A()
    requires P_1 ∨ P_2 ∨ ··· ∨ P_m
    ensures Q_1 ∧ Q_2 ∧ ··· ∧ Q_n
```

This will eventually lead to a conjecture of the form

$$P_1 \vee P_2 \vee \cdots \vee P_m \Rightarrow (Q_1' \wedge \cdots \wedge Q_n'),$$

(where each Q' is the weakest precondition of the corresponding Q with respect to the body of A). This, in turn, will require testing the consistency of the query

$$(P_1 \vee P_2 \vee \cdots \vee P_m) \wedge (\neg Q_1' \vee \cdots \vee \neg Q_n').$$

In this situation, it is heuristically preferable to split on the clause containing the Q''s, rather than on the precondition. That is to say, if there are multiple postconditions to be proved, and multiple cases in the precondition, it is generally faster to prove the postconditions one at a time than to explore the various cases of the precondition one at a time. (More generally, the procedure precondition could contain many disjunctions, and even it contains none, disjunctions will appear in the antecedent of the verification condition if there are conditionals in the procedure body.)

A similar situation arises in proving the precondition for an internally called procedure, in case the precondition is a conjunction. It is heuristically preferable to prove the conjuncts one at a time, rather than to perform any other case splits that may occur in the verification condition.

In order to allow the verification condition generator to give Simplify hints about which case splits to favor, we simply adopt the policy of favoring splits in Q rather than in P when we are proving a formula of the form $P \Rightarrow Q$. This preference is inherited when proving the various elements of a conjunction; for example, in
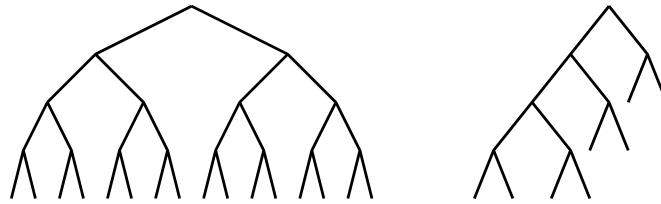
$P \Rightarrow ((Q_1 \Rightarrow R_1) \wedge (Q_2 \Rightarrow R_2))$, case splits in R_i will be favored over case splits in Q_i or in P.

We implement this idea by adding a Boolean *goal* property to literals and to clauses. When a goal proxy for $P \Rightarrow Q$ is denied, the proxy for P is asserted, the proxy for Q is denied, and the proxy for Q (only) is made a goal. When a goal proxy for a conjunction is denied, producing a clause of two denied proxy literals, the clause and each of its literals become goals. The proxy for the initial query is also given the goal property. When choosing a case split, Simplify favors goal clauses.

Instead of treating implication asymmetrically, it would have been possible to alter the input syntax to allow the user to indicate the goal attribute in a more flexible way, but we have not done so.

3.6. SCORING CLAUSES.   In our applications, we find that an unsatisfiable clause set frequently contains many irrelevant clauses: its unsatisfiability follows from a small number of relevant clauses. In such a case, if Simplify is lucky enough to split on the relevant clauses first, then the proof search will go quickly. But if Simplify is unlucky enough to split on the irrelevant clauses before splitting on the relevant ones, then the proof search will be very slow.

To deal with this problem, we associate a *score* with each clause. When choosing a case split, we favor clauses with higher scores. (This preference for high-scoring clauses is given less priority than the preference for goal clauses.) Each time a contradiction leads Simplify to backtrack, Simplify increments the score of the last clause split on.

Figure 1 shows how this heuristic works in a particularly simple case, where there are n binary clauses, only one of which is relevant. Let the clauses of the context be

$$(P_1 \vee Q_1) \wedge (P_2 \vee Q_2) \wedge \cdots \wedge (P_n \vee Q_n)$$

and suppose that only the last clause is relevant. That is, each of P_n and Q_n is inconsistent with the context, and none of the other literals have any relevant effect on the context at all. Without scoring (and temporarily ignoring width reduction (§3.2)), if Simplify considers the clauses in the unlucky order in which they are listed, the search tree has 2^n leaves, as illustrated in the left of the figure. With scoring, the proof tree has only $2n$ leaves, as illustrated in the right of the figure. Since asserting a literal from an irrelevant clauses never leads to a contradiction, the scores of these clauses will never be incremented. When the relevant clause $P_n \vee Q_n$ is considered, its score will be incremented by 2. For the rest of the proof, the relevant clause will be favored over all irrelevant clauses.

The scoring heuristic is also helpful if there is more than one relevant clause. The reader may wish to work out the proof tree in the case that two binary clauses are relevant (in the sense that the four possible ways of choosing one literal from each of the relevant clauses are all inconsistent) and $n - 2$ are irrelevant. In general, if there are k relevant binary clauses and $n - k$ irrelevant binary clauses, the scoring heuristic produces a search tree with at most $n2^k$ leaves.

So much for the basic idea of scoring. In the actual implementation, there are many details that need to be addressed. We will spare the reader most of them, but mention two that are of some importance.

First, in order to be of any use, incrementing the score of a clause must not be undone by *Pop*. However, scores do need to be reset periodically, since different clauses are often relevant in different parts of the proof. Simplify resets scores whenever it backtracks from a case split on a nongoal clause and the previous case split on the current path (§3.1) was on a goal clause. When Simplify resets scores, it renormalizes all scores to be in the range zero to one, causing high scoring clauses to retain some advantage but giving low scoring clauses a chance to catch up.

Second, there are interactions between scoring and lazy CNF. When a clause C contains a proxy literal P whose assertion leads to the introduction of another clause D, then C is referred to as the *parent clause* of D. Suppose the child clause D is relevant and acquires a high score. When Simplify backtracks high up in the proof tree, above the split on the parent clause C, the clause D will no longer be present. The only way to reintroduce the useful clause D is to split again on C. We take the view that D's high score should to some extent lead us to favor splitting on C, thus reintroducing D. Therefore, when Simplify increases the score of a clause, it also increases (to a lesser extent) the scores of the parent and grandparent clauses. Of course, the score for the clause D is not reset each time its proxy causes it to be introduced.

3.7. USING THE SUBSUMPTION HEURISTIC FOR PROXY LITERALS.

In Section 3.4, we noted that the subsumption heuristic (§3.2) may interact poorly with lazy CNF. Specifically, applying the subsumption heuristic to proxy literals could reintroduce the exponential mixing that lazy CNF was designed to avoid. In fact, Simplify uses a modified version of the subsumption heuristic that regains some of the benefits without the risk of reintroducing exponential mixing.

Suppose Simplify does a case split on a proxy literal l of a clause c. After backtracking from the case where l holds and deleting l from the clause c, it adds $\neg l$ to the literal set, but does not add the expansion of $\neg l$ to the clause set. Since the expansion of l is not added to the clause set, it cannot be a source of exponential mixing. However, if l is a proxy for a repeated subformula, other clauses containing l or $\neg l$ may occur in the clause set, and the presence of $\neg l$ in the literal set will enable width reduction or clause elimination (§3.2).

A subtlety of this scheme is that Simplify must keep track of whether each proxy has had its expansion added to the clause set on the current path. If a "never-expanded" proxy literal l in the literal set is used to eliminate a clause $(\cdots \vee l \vee \cdots)$ from the clause set, the expansion of l must be added to the clause set at that point. Otherwise, Simplify might find a "satisfying assignment" (§2) that does not actually satisfy the query.

With the modification described here, *Sat* no longer maintains the invariant that all proxy literals in *lits* are redundant. We leave it as an exercise for the reader to demonstrate the correctness of the modified algorithm.

4. *Domain-Specific Decision Procedures*

In this section, we show how to generalize the propositional theorem-proving methods described in the previous section to handle the functions and relations of Simplify's built-in theory.

The *Sat* algorithm requires the capability to test the consistency of a set of literals. Testing the consistency of a set of propositional literals is easy: the set is consistent unless it contains a pair of complementary literals. Our strategy for handling formulas involving arithmetic and equality is to retain the basic *Sat* algorithm described above, but to generalize the consistency test to check the consistency of sets of arbitrary literals. That is, we implement the satisfiability interface from Section 3.1:

> **var** *refuted*: **boolean**
> **proc** *AssertLit*(*L* : *Literal*)
> **proc** *Push*()
> **proc** *Pop*()

but with *L* ranging over the literals of Simplify's predefined theory. The implementation is sound but is incomplete for the linear theory of integers and for the theory of nonlinear multiplication. As we shall see, the implementation is complete for a theory which is, in a natural sense, the combination of the theory of equality (with uninterpreted function symbols) and the theory of rational linear arithmetic.

Two important modules in Simplify are the *E-graph* module and the *Simplex* module. Each module implements a version of *AssertLit* for literals of a particular theory: the E-graph module asserts literals of the theory of equality with uninterpreted function symbols; the Simplex module asserts literals of rational linear arithmetic. When the *AssertLit* method of either module detects a contradiction, it sets the global *refuted* bit. Furthermore each *AssertLit* routine must push sufficient information onto the undo stack (§3.1) so that its effects can be undone by *Pop*. A fine point: Instead of using the global undo stack, Simplify's theory modules actually maintain their own private undo stacks and export *Push* and *Pop* procedures, which are called (only) by the global *Push* and *Pop*. We're not sure we'd do it this way if we had it to do over. In any case, in this paper, *Pop* always means to pop the state of the entire context.

Because the context may include literals from both theories, neither satisfiability procedure by itself is sufficient. What we really want is a satisfiability procedure for the combination of the theories. The first decision procedure for linear arithmetic combined with function symbols was invented by Robert Shostak [Shostak 1979], but this method was specific to these two theories. Simplify employs a general method for combining decision procedures, known as *equality sharing*.

The equality sharing technique was introduced in Nelson's Ph.D. thesis [Nelson 1979]. Two more modern expositions of the method, including proofs of correctness, are by Tinelli and Harandi [1996] and by Nelson [1983]. In this technique, a collection of decision procedures work together on a conjunction of literals; each working on one logical theory. If each decision procedure is complete, and the

individual theories are "convex" (a notion defined in the papers just cited), and if each decision procedure shares with the others any equality between variables that is implied by its portion of the conjunction, then the collective effort will also be complete. The theory of equality with uninterpreted function symbols is convex, and so is the theory of linear rational inequalities, so the equality sharing technique is appropriate to use with the E-graph and Simplex modules.

We describe equality sharing in Section 4.1 and give high level descriptions of the E-graph and Simplex modules in Sections 4.2 and 4.3. Sections 7 and 8 provide more detailed discussions of the implementations, including undoing among other topics. Sections 4.4 and 4.5 give further practical details of the implementation of equality sharing. Section 4.6 describes modifications to the *Refine* procedure enabled by the non-propositional literals of Simplify's built-in theory. Section 4.7 describes the built-in theory of partial orders.

4.1. EQUALITY SHARING.   For a logical theory $\mathcal{T}$, a $\mathcal{T}$-*literal* is a literal whose function and relation symbols are all from the language of $\mathcal{T}$.

The *satisfiability problem* for a theory $\mathcal{T}$ is the problem of determining the satisfiability of a conjunction of $\mathcal{T}$-literals (also known as a $\mathcal{T}$-*monome*).

The satisfiability problem for a theory is the essential computational problem of implementing the satisfiability interface (§3.1) for literals of that theory.

Example. Let $\mathcal{R}$ be the additive theory of the real numbers, with function symbols

$$+, -, 0, 1, 2, 3, \ldots$$

and relation symbols

$$=, \leq, \geq$$

and the axioms of an ordered field. Then the satisfiability problem for $\mathcal{R}$ is essentially the linear programming satisfiability problem, since each $\mathcal{R}$-literal is a linear equality or inequality (§2).

If $\mathcal{S}$ and $\mathcal{T}$ are theories, we define $\mathcal{S} \cup \mathcal{T}$ as the theory whose relation symbols, function symbols, and axioms are the unions of the corresponding sets for $\mathcal{S}$ and for $\mathcal{T}$.

Example. Let $\mathcal{E}$ be the theory of equality with the single relation symbol

$$=$$

and an adequate supply of "uninterpreted" function symbols

$$f, g, h, \ldots$$

Then the satisfiability problem for $\mathcal{R} \cup \mathcal{E}$ includes, for example, the problem of determining the satisfiability of

$$
\begin{aligned}
& f(f(x) - f(y)) \neq f(z) \\
\wedge\ & x \leq y \\
\wedge\ & y + z \leq x \\
\wedge\ & 0 \leq z.
\end{aligned}
\tag{4}
$$

The separate satisfiability problems for $\mathcal{R}$ and $\mathcal{E}$ were solved long ago, by Fourier and Ackerman, respectively. But the combined problem was not considered until it became relevant to program verification.

Equality sharing is a general technique for solving the satisfiability problem for the theory $\mathcal{S} \cup \mathcal{T}$, given solutions for $\mathcal{S}$ and for $\mathcal{T}$, and assuming that (1) $\mathcal{S}$ and $\mathcal{T}$ are both first order theories with equality and (2) $\mathcal{S}$ and $\mathcal{T}$ have no other common function or relation symbol besides $=$.

The technique produces efficient results for cases of practical importance, including $\mathcal{R} \cup \mathcal{E}$.

By way of example, we now describe how the equality-sharing procedure shows the inconsistency of the monome (4) above.

First, a definition: in a term or atomic formula of the form $f(\ldots, g(\ldots), \ldots)$, the occurrence of the term $g(\ldots)$ is called *alien* if the function symbol g does not belong to the same theory as the function or relation symbol f. For example, in (4), $f(x)$ occurs as an alien in $f(x) - f(y)$, because $-$ is an $\mathcal{R}$ function but f is not.

To use the postulated satisfiability procedures for $\mathcal{R}$ and $\mathcal{E}$, we must extract an $\mathcal{R}$-monome and an $\mathcal{E}$-monome from (4). To do this, we make each literal homogeneous (alien-free) by introducing names for alien subexpressions as necessary. Every literal of (4) except the first is already homogeneous. To make the first homogeneous, we introduce the name g_1 for the subexpression $f(x) - f(y)$, g_2 for $f(x)$, and g_3 for $f(y)$. The result is that (4) is converted into the following two monomes:

$\mathcal{E}$-monome	$\mathcal{R}$-monome
$f(g_1) \neq f(z)$	$g_1 = g_2 - g_3$
$f(x) = g_2$	$x \leq y$
$f(y) = g_3$	$y + z \leq x$
	$0 \leq z$

This homogenization is always possible, because each theory includes an inexhaustible supply of names and each theory includes equality.

In this example, each monome is satisfiable by itself, so the detection of the inconsistency must involve interaction between the two satisfiability procedures. The remarkable news is that a particular limited form of interaction suffices to detect inconsistency: each satisfiability procedure must detect and propagate to the other any equalities between variables that are implied by its monome.

In this example, the satisfiability procedure for $\mathcal{R}$ detects and propagates the equality $x = y$. This allows the satisfiability procedure for $\mathcal{E}$ to detect and propagate the equality $g_2 = g_3$. Now the satisfiability procedure for $\mathcal{R}$ detects and propagates the equality $g_1 = z$, from which the satisfiability procedure for E detects the inconsistency.

If we had treated the free variables "x" and "y" in the example above as Skolem constants "$x()$" and "$y()$", then the literal "$x \leq y$" would become "$x() \leq y()$", which would be homogenized to something like "$g_4 \leq g_5$" where $g_4 = x()$ and $g_5 = y()$ would be the defining literals for the new names. The rule that equalities between variables must be propagated would now apply to the g's even though $x()$ and $y()$ would not be subject to the requirement. So the computation is essentially the same regardless of whether x and y are viewed as variables or as Skolem constants.

Implementing the equality-sharing procedure efficiently is surprisingly subtle. It would be inefficient to create explicit symbolic names for alien terms and to introduce the equalities defining these names as explicit formulas. Sections 4.4 and 4.5 explain the more efficient approach used by Simplify.

The equality-sharing method for combining decision procedures is often called the "Nelson–Oppen method" after Greg Nelson and Derek Oppen, who first implemented the method as part of the Stanford Pascal Verifier in 1976–79 in a MACLisp program that was also called Simplify but is not to be confused with the Simplify that is the subject of this article. The phrase "Nelson–Oppen method" is often used in contrast to the "Shostak method" invented a few years later by Rob Shostak at SRI [Shostak 1984]. Furthermore, it is often asserted that the Shostak method is "ten times faster than the Nelson–Oppen method".

The main reason that we didn't use Shostak's method is that we didn't (and don't) understand it as well as we understand equality sharing. Shostak's original paper contained several errors and ambiguities. After Simplify's design was settled, several papers appeared correcting and clarifying Shostak's method [Ruess and Shankar 2001; Barrett et al. 2002b; Barrett 2002]. The consensus of these papers seems to be that Shostak's method is not so much an independent combining method but a refinement of the Nelson–Oppen method for the case when the theory admits a solver and a canonizer. A number of other recent papers refine and discuss Shostak's method [Shankar and Ruess 2002; Conchon and Krstić 2003; Krstić and Conchon 2003; Ganzinger 2002; Ganzinger et al. 2004].

We still lack the firm understanding we would want to have to build a tool based on Shostak's ideas, but we do believe these ideas could be used to improve performance. We think it is an important open question how much improvement there would be. It is still just possible to trace the history of the oft-repeated assertion that Shostak's method is ten times faster than the Nelson–Oppen method. The source seems to be a comparison done in 1981 by Leo Marcus at SRI, as reported by Steve Crocker [Marcus 1981; Crocker 1988]. But Marcus's benchmarks were tiny theorems that were not derived from actual program checking problems. In addition, it is unclear whether the implementations being compared were of comparable quality. So we do not believe there is adequate evidence for the claimed factor of ten difference. One obstacle to settling this important open question is the difficulty of measuring the cost of the combination method separately from the many other costs of an automatic theorem prover.

4.2. THE E-GRAPH MODULE.   We now describe Simplify's implementation of the satisfiability interface for the theory of equality, that is, for literals of the forms $X = Y$ and $X \neq Y$, where X and Y are terms built from variables and applications of uninterpreted function symbols. In this section, we give a high-level description of the satisfiability procedure; Section 7 contains a more detailed description.

We use the facts that equality is an equivalence relation—that is, that it is reflexive, symmetric, and transitive—and that it is a congruence—that is, if x and y are equal, then so are $f(x)$ and $f(y)$ for any function f.

Before presenting the decision procedure, we give a simple example illustrating how the properties of equality can be used to test (and in this case, refute) the satisfiability of a set of literals. Consider the set of literals

$$
\begin{array}{ll}
1. & f(a, b) = a \\
2. & f(f(a, b), b) = c \\
3. & g(a) \neq g(c)
\end{array}
\tag{5}
$$

It is easy to see that this set of literals is inconsistent:

4. $f(f(a, b), b) = f(a, b)$ (from 1, $b = b$, and congruence)
5. $f(a, b) = c$ (from 4, symmetry on 4, 2, and transitivity)
6. $a = c$ (from 5, 1, symmetry on 1, and transitivity)
7. $g(a) = g(c)$ (from 6 and congruence), which contradicts 3.

We now briefly describe the data structures and the algorithms used by Simplify to implement reasoning of the kind used in the example above. Section 7 provides a more detailed description.

A *term DAG* is a vertex-labeled directed oriented acyclic multigraph, whose nodes represent ground terms (§2). By *oriented* we mean that the edges leaving any node are ordered. If there is an edge from u to v, we call u a *parent* of v and v a *child* of u. We write $\lambda(u)$ to denote the label of u, we write $degree(u)$ to denote the number of edges from u, and we write $u[i]$ to denote the ith child of u, where the children are ordered according to the edge ordering out of u. We write *children*$[u]$ to denote the sequence $u[1], \ldots, u[degree(u)]$. By a multigraph, we mean a graph possibly with multiple edges between the same pairs of nodes (so that possibly $u[i] = u[j]$ for $i \neq j$). A term $f(t_1, \ldots, t_n)$ is represented by a node u if $\lambda(u) = f$ and *children*$[u]$ is a sequence $v_1, \ldots, v_n$ where each v_i represents t_i.

The term DAG used by the satisfiability procedure for $\mathcal{E}$ represents ground terms only. We will consider explicit quantifiers in Section 5.

Given an equivalence relation R on the nodes of a term DAG, we say that two nodes u and v are *congruent under R* if $\lambda(u) = \lambda(v)$, $degree(u) = degree(v)$, and for each i in the range $1 \leq i \leq degree(u)$, $R(u[i], v[i])$. The set of nodes congruent to a given node is a called a *congruence class*. We say that equivalence relation R is *congruence-closed* if any two nodes that are congruent under R are also equivalent under R. The *congruence closure* of a relation R on the nodes of a term DAG is the smallest congruence-closed equivalence relation that extends R.

An *E-graph* is a data structure that includes a term DAG and an equivalence relation on the term DAG's nodes (called *E-nodes*). The equivalence relation relates a node u to a node v if and only if the terms represented by u and v are guaranteed to be equal in the context represented by the E-graph.

From now on, when we say that an E-node u *represents* a term $f(t_1, \ldots, t_n)$, we mean that the label of u is f and that each child $u[i]$ of u is equivalent to some E-node that represents t_i. That is, represents means "represents up to congruence".

We can now describe the basic satisfiability procedure for $\mathcal{E}$. To test the satisfiability of an arbitrary $\mathcal{E}$-monome M, we proceed as follows: First, we construct an E-graph whose term DAG represents each term in M and whose equivalence relation relates $node(T)$ to $node(U)$ whenever M includes the equality $T = U$. Second, we close the equivalence relation under congruence by repeatedly merging the equivalence classes of any nodes that are congruent but not equivalent. Finally, we test whether any literal of M is a distinction $T \neq U$ where $node(T)$ and $node(U)$ are equivalent. If so, we report that M is unsatisfiable; otherwise, we report that M is satisfiable.

Figure 2 shows the operation of this algorithm on the example (5) above. Note that the variables a, b, and c are represented by leaf E-nodes of the E-graph. As explained near the end of Section 2, the equivalence underlying the Skolemization technique implies that it doesn't matter whether we treat a, b, and c as variables or
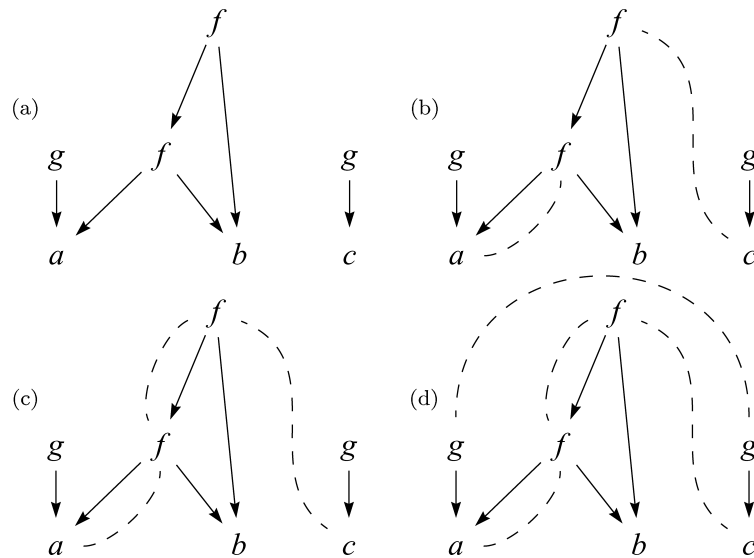
FIG. 2. Application of the congruence closure to example (5). (a) A term DAG for the terms in (5). (b) The E-graph whose equivalences (shown by dashed lines) correspond to the equalities in (5). (c) $node(f(f(a, b), b))$ and $node(f(a, b))$ are congruent in (b); make them equivalent. (d) $node(g(a))$ and $node(g(c))$ are congruent in (c); make them equivalent. Since $g(a)$ and $g(c)$ are distinguished in (5) but equivalent in (d), (5) is unsatisfiable.

as symbolic literal constants. The E-graph pictured in Figure 2 makes them labeled nodes with zero arguments, that is, symbolic literal constants.

This decision procedure is sound and complete. It is sound since, by the construction of the equivalence relation of the E-graph, two nodes are equivalent in the congruence closure only if they represent terms whose equality is implied by the equalities in M together with the reflexive, symmetric, transitive, and congruence properties of equality. Thus, the procedure reports M to be unsatisfiable only if it actually finds two nodes $node(T_1)$ and $node(T_2)$ such that both $T_1 = T_2$ and $T_1 \neq T_2$ are consequences of M in the theory $\mathcal{E}$. It is complete since, if it reports monome M to be satisfiable, the equivalence classes of the E-graph provide a model that satisfies the literals of M as well as the axioms of equality. (The fact that the relation is congruence-closed ensures that the interpretations of function symbols in the model are well defined.)

The decision procedure for $\mathcal{E}$ is easily adapted to participate in the equality-sharing protocol: it is naturally incremental with respect to asserted equalities (to assert an equality $T = U$ it suffices to merge the equivalence classes of $node(T)$ and $node(U)$ and close under congruence) and it is straightforward to detect and propagate equalities when equivalence classes are merged.

To make the E-graph module incremental with respect to distinctions (§2), we also maintain a data structure representing a set of forbidden merges. To assert $x \neq y$, we forbid the merge of x's equivalence class with y's equivalence class by adding the pair (x, y) to the set of forbidden merges. The set is checked before performing any merge, and if the merge is forbidden, *refuted* is set.

In Section 7, we describe in detail an efficient implementation of the E-graph, including non-binary distinctions and undoing. For now, we remark that, by using

methods described by Downey et al. [1980], our implementation guarantees that incrementally asserting the literals of any $\mathcal{E}$-monome (with no backtracking) requires a worst-case total cost of $O(n \log n)$ expected time, where n is the print size of the monome. We also introduce here the *root* field of an E-node: $v.root$ is the canonical representative of v's equivalence class.

4.3. THE SIMPLEX MODULE.   Simplify's *Simplex module* implements the satisfiability interface for the theory $\mathcal{R}$. The name of the module comes from the Simplex algorithm, which is the central algorithm of its *AssertLit* method. The module is described in some detail in Section 8. For now, we merely summarize its salient properties.

The Simplex method is sound and complete for determining the satisfiability over the rationals of a conjunction of linear inequalities. Simplify also employs some heuristics that are sound but incomplete for determining satisfiability over the integers.

The space required is that for a matrix—the *Simplex tableau*—with one row for every inequality (§2) and one column for every Simplex unknown. The entries in the matrix are integer pairs representing rational numbers.

In the worst case, the algorithm requires exponential time, but this worst case is very unlikely to arise. In practice, the per-assertion cost is a small number of pivots of the tableau, where the cost of a pivot is proportional to the size of the tableau (see Section 9.15).

In our applications the unknowns represent integers. We take advantage of this to eliminate strict inequalities, replacing $X < Y$ by $X \leq Y - 1$. This partially compensates for the fact that the Simplex algorithm detects unsatisfiability over the rationals rather than over the integers. Two other heuristics, described in Section 8, offer additional compensation, but Simplify is not complete for integer linear arithmetic.

4.4. ORDINARY THEORIES AND THE SPECIAL ROLE OF THE E-GRAPH.   In Section 4.1, we described the equality-sharing procedure as though the roles of the participating theories were entirely symmetric. In fact, in the implementation of Simplify, the theory of equality with uninterpreted function symbols plays a special role. Its decision procedure, the E-graph module, serves as a central repository representing all ground terms in the conjecture.

Each of the other built-in theories is called an *ordinary* theory.

For each ordinary theory T, Simplify includes an incremental, resettable decision procedure for satisfiability of conjunctions of T-literals (§4.1). We use the name T both for the theory and for this module. In this section we describe the interface between an ordinary theory and the rest of Simplify. This interface is very like the satisfiability interface of Section 3.1, but with the following differences:

The first difference is that the module for an ordinary theory T declares a type $T.Unknown$ to represent unknowns of T. In order to maintain the association between E-nodes and unknowns, for each ordinary theory T, each E-node has a $T\_unknown$ field whose value is a $T.Unknown$ (or **nil**). It may seem wasteful of space to include a separate pointer field in each E-node for each ordinary theory, but the number of ordinary theories is not large (in the case of Simplify, the number is two), and there are straightforward techniques (implemented in Simplify but not described in this paper) that reduce the space cost in practice.

Each ordinary theory T introduces into the class $T.Unknown$ whatever fields its satisfiability procedure needs, but there is one field common to all the various $T.Unknown$ classes: the *enode* field, which serves as a kind of inverse to the $T\_unknown$ field. More precisely:

(1) for any E-node e, if $e.T\_unknown \neq$ **nil**, then $e.T\_unknown.enode$ is equivalent to e, and

(2) for any $T.Unknown$ u, if $u.enode \neq$ **nil**, then $u.enode.T\_unknown = u$.

An unknown u is *connected* if $u.enode \neq$ **nil**.

Each ordinary theory T must implement the following method for generating an unknown and connecting it to an E-node.

> **proc** $T.UnknownForEnode(e : E\text{-}node) : Unknown$;
> /∗ Requires that $e.root = e$. Returns $e.T\_unknown$ if $e.T\_unknown \neq$ **nil**. Otherwise sets $e.T\_unknown$ to a newly-allocated unconstrained unknown (with *enode* field initialized to e) and returns it. ∗/

The second difference is that the literals passed to *AssertLit* are not propositional unknowns but literals of T. In particular, *AssertLit* must accept literals of the following kinds:

$$u_1 = u_2$$
$$u_0 = F(u_1, \ldots, u_n) \quad \text{for each } n\text{-ary function symbol } F \text{ of } T$$
$$R(u_1, \ldots, u_n) \quad \text{for each } n\text{-ary relation symbol } R \text{ of } T$$
$$\neg R(u_1, \ldots, u_n) \quad \text{for each } n\text{-ary relation symbol } R \text{ of } T$$

where the u's are unknowns of T. There must be procedures for building these literals from unknowns, but we will not describe those procedures further here.

We introduce the abstract variable $T.Asserted$ to represent the conjunction of currently asserted T literals.

The third difference is that *AssertLit* must propagate equalities as well as check consistency. We introduce the abstract variable $T.Propagated$ to represent the conjunction of equalities currently propagated from T. The T module is responsible for maintaining the invariant:

Invariant (PROPAGATION FROM T). For any two connected unknowns u and v, the equality $u = v$ is implied by *Propagated* iff it is implied by *Asserted*.

In summary, the specification for $T.AssertLit$ is:

> **proc** $T.AssertLit(L : T\text{-}Literal)$;
> /∗ If L is consistent with $T.Asserted$, set $T.Asserted$ to $L \wedge T.Asserted$, propagating equalities as required to maintain PROPAGATION FROM T. Otherwise set *refuted* to **true**. ∗/

Each ordinary theory T can assume that the E-graph module maintains the following invariant:

Invariant (PROPAGATION TO T). Two equivalent E-nodes have non-nil $T\_unknown$ fields if and only if these two $T.Unknown$'s are equated by a chain of currently propagated equalities from the E-graph to the T module.

Section 7 describes the E-graph code that maintains this invariant.

Note that, while $T.Asserted$ may imply a quadratic number of equalities between T-unknowns, at most $n - 1$ of these (where n is the number of T-unknowns) need to be propagated on any path in order to maintain PROPAGATION FROM T.

Similarly, at most $n - 1$ equalities need be propagated from the E-graph to T in order to maintain PROPAGATION TO T.

A fine point: It may be awkward for a theory to deal with an incoming equality assertion while it is in the midst of determining what equalities to propagate as a result of some previous assertion. To avoid this awkwardness, propagated equalities (both from and to T) are not asserted immediately but instead are put onto a work list. Simplify includes code that removes and asserts equalities from the work list until it is empty or the current case is refuted.

4.5. CONNECTING THE E-GRAPH WITH THE ORDINARY THEORIES. A literal of the conjecture may be inhomogeneous—that is, it may contain occurrence of functions and relations of more than one theory. In our initial description of equality sharing in Section 4.1, we dealt with this problem by introducing new variables as names for the alien (§4.1) terms and defining the names with additional equalities. This is a convenient way of dealing with the problem in the language of first order logic, but in the actual implementation there is no need to introduce new variables.

Instead, Simplify creates E-nodes not only for applications of uninterpreted function symbols, but for all ground terms in the conjecture. For each E-node that is relevant to an ordinary theory T because it is an application of or an argument to a function symbol of T, Simplify allocates a *T.Unknown* and links it to the term's E-node. More precisely, for a term $f(t_1, \ldots, t_k)$ where f is a function symbol of T, the E-nodes representing the term and its arguments are associated with *T.Unknown*'s and the relation between these $k + 1$ unknowns is represented by an assertion of the appropriate T-literal.

For example, if p is the E-node for an application of the function symbol $+$, and q and r are the two children of p, then the appropriate connections between the E-graph and the Simplex module are made by the following fragment of code:

```
var u := Simplex.UnknownForEnode(p.root),
    v := Simplex.UnknownForEnode(q.root),
    w := Simplex.UnknownForEnode(r.root) in
  Simplex.AssertLit(u = v + w)
```

There are a variety of possible answers to the question of exactly when these connections are made. A simple answer would be to make the connections eagerly whenever an E-node is created that represents an application of a function of an ordinary theory. But we found it more efficient to use a lazier strategy in which the connections are made at the first point on each path where the *Sat* algorithm asserts a literal in which the function application occurs.

Not only the function symbols but also the relation symbols of an ordinary theory give rise to connections between E-nodes and unknowns. For each expression of the form $R(t_1, \ldots, t_k)$, where R is a relation of an ordinary theory T, Simplify creates a data structure called an AF (atomic formula (§2)). The AF a for $R(t_1, \ldots, t_k)$ is such that *AssertLit(a)* asserts $R(u_1, \ldots, u_n)$ to T, and *AssertLit($\neg a$)* asserts $\neg R(u_1, \ldots, u_n)$ to T, where u_i is the T-unknown connected to the E-node for t_i

The use of unknowns to connect the E-graph with the Simplex tableau was described in Nelson's thesis [Nelson 1981, Sec. 13].

Building the collection of E-nodes and unknowns that represents a term is called *interning* the term.

An implementation note: The AF for $R(t_1, \ldots, t_k)$ may contain pointers either to the E-nodes for the t's or to the corresponding unknowns (u's), depending on whether the connections to T are made lazily or eagerly. The representation of a literal is an AF paired with a *sense* Boolean, indicating whether the literal is positive or negative. The generic AF has an *Assert* method which is implemented differently for different subtypes of AF each of which corresponds to a different relation R. To assert a literal, its AF's *Assert* method is called with the sense boolean as a parameter.

It is heuristically desirable to canonicalize (§3.3) AF's as much as possible, so that, for example, if the context contains occurrences of $x < y$, $y > x$, $\neg x \geq y$, and $\neg y \leq x$, then all four formulas are canonicalized to the same AF. The E-graph module exploits symmetry and the current equivalence relation to canonicalize equalities and binary distinctions (§2), but Simplify leaves it to each ordinary theory to expend an appropriate amount of effort in canonicalizing applications of its own relation symbols.

In accordance with our plan to distinguish functions from relations, we did not make AF a subtype of *E-node*: the labels in the E-graph are always function symbols, never relations. In retrospect, we suspect this was a mistake. For example, because of this decision, the canonicalization code for AF's in an ordinary theory must duplicate the functionality that the E-graph module uses to produce canonical E-nodes for terms. An even more unpleasant consequence of this decision is that matching triggers (see Section 5.1) cannot include relation symbols. At one point in the ESC project, this consequence (in the particular case of binary distinctions) become so debilitating that we programmed an explicit exception to work around it: we reserved the quasi-relation symbol **neq** and modified the assert method for equality AF's so that $\textbf{neq}(t, u) = @\textbf{true}$ is asserted whenever $t = u$ is denied.

4.6. WIDTH REDUCTION WITH DOMAIN-SPECIFIC LITERALS. The domain-specific decision procedures create some new opportunities for the *Refine* procedure to do width reduction and clause elimination (§3.2). Suppose l is a literal in some clause c in the clause set *cls*. The version of *Refine* in Section 3.2 deletes the c from *cls* (clause elimination) if *lits*—viewed a a set of literals—contains l, and it deletes l from c (width reduction) if *lits* contains $\neg l$. In fact deleting l from *cls* will leave the meaning of the context unchanged if *lits*—viewed as a conjunction of literals—implies l (equivalently, if $\neg l$ is inconsistent with *lits*). Similarly, deleting l from c will leave the meaning of the context unchanged if l is inconsistent with (the conjunction of) *lits* (equivalently, if *lits* implies $\neg l$).

For a consistent literal sets containing only propositional variables and their negations, as in Section 3, containment ($l \in \textit{lits}$) and implication ($[\textit{lits} \Rightarrow l]$) are equivalent. For the larger class of literals of Simplify's built-in theory, this equivalence no longer holds. For example, the conjunction $x < y \wedge y < z$ implies the literal $x < z$, even though the set $\{x < y, y < z\}$ does not contain the literal $x < z$.

Since we have a decision procedure for equalities, distinctions, and inequalities that is complete, incremental, and resettable, it is easy to write a procedure to test a literal for consistency with the current literal set. Here are the specification and implementation of such a procedure:

```
proc Implied(L : Literal): boolean
/* Returns true if and only if [lits ⇒ L]. */
```

```
proc Implied(L : Literal) ≡
    Push();
    AssertLit(¬ L);
    if refuted then
        Pop(); return true
    else
        Pop(); return false
    end
end
```

We refer to this method of testing implication by *lits* as *plunging* on the literal. Plunging is somewhat expensive, because of the overhead in *Push*, *Pop*, and especially *AssertLit*.

On the other hand, we can test for membership of a literal (or its complement) in *lits* very cheaply by simply examining the sense of the literal and the status (§3.2) field of the literal's AF, but this *status test* is less effective than plunging at finding opportunities for width reduction and clause elimination. The effectiveness of the status test is increased by careful canonicalization of atomic formulas into AF's.

For literals representing equalities and distinctions, Simplify includes tests (the *E-graph tests*) that are more complete than the status test but less expensive than plunging: the E-graph implies the equality $T = U$ if the E-nodes for T and U are in the same equivalence class, and it implies the distinction $T \neq U$ if (albeit not only if) the equivalence classes of the E-nodes for T and U have been forbidden to be merged.

To compare the three kinds of tests, consider a context in which the following two literals have been asserted:

$$i = j, \quad f(j) \neq f(k).$$

Then

—$j = i$ would be inferred by the status test because $i = j$ and $j = i$ are canonicalized identically,

—$f(i) \neq f(k)$ would be inferred by the E-graph test since $f(i)$ and $f(j)$ are congruent (§4.2), hence equivalent but not by the status test if $f(i) \neq f(k)$ was canonicalized before $i = j$ was asserted, and

—$j \neq k$ would be inferred by plunging (since a trial assertion would quickly refute $j = k$) but would not by inferred either the status test or the E-graph test.

In an early version of Simplify, we never did a case split without first applying the plunging version of *Refine* to every clause. We found this to be too slow. In the current version of Simplify, we apply the plunging version of *Refine* to each non-unit clause produced by matching (see Section 5.2), but we do this just once, immediately after the match is found. On the other hand, we continue to use E-graph tests aggressively: we never do a case split without first applying the E-graph test version of *Refine* to every clause.

4.7. THE THEORY OF PARTIAL ORDERS. If f and g are binary quasirelations, the syntax

$$(\text{ORDER } f \ g)$$

is somewhat like a higher order atomic formula that asserts that f and g are the irreflexive and reflexive versions, respectively, of a partial order. (We write "somewhat like" because Simplify's logic is first order and this facility's implementation is more like a macro than a true higher-order predicate.)

Each assertion of an application of ORDER dynamically creates a new instance of a prover module whose satisfiability procedure performs transitive closure to reason about assertions involving the two quasi-relations.

The orders facility is somewhat *ad hoc*, and we will not describe all its details in this article. The interface to the dynamically created satisfiability procedures is mostly like the interface described in Sections 4.4 and 4.5, but the procedures propagate not just equalities but also ordering relations back to the E-graph, where they can be used by the matcher to instantiate universally quantified formulas as described in the next section. For example, this allows Simplify to infer

$$a \text{ LT } b \;\Rightarrow\; R(a, b)$$

from

$$(\text{ORDER LT LE}) \wedge (\forall x, y : x \text{ LE } y \;\Rightarrow\; R(x, y)).$$

5. *Quantifiers*

So far, we have ignored quantifiers. But a treatise on theorem-proving that ignores quantifiers is like a treatise on arithmetic that ignores multiplication: quantifiers are near the heart of all the essential difficulties.

With the inclusion of quantifiers, the theoretical difficulty of the theorem-proving problem jumps from *NP*-complete to undecidable (actually, semidecidable: an unbounded search of all proofs will eventually find a proof if one exists, but no bounded search will do so). Much of the previous work in automatic theorem-proving (as described for example in Donald Loveland's book [Loveland 1978]) has concentrated on strategies for handling quantifiers that are *complete*, that is, that are guaranteed in principle eventually to find a proof if a proof exists. But for our goal, which is to find simple proofs rapidly when simple proofs exist, a complete search strategy does not seem to be essential.

5.1. OVERVIEW OF MATCHING AND TRIGGERS. Semantically, the formula $(\forall x_1, \ldots, x_n : P)$ is equivalent to the infinite conjunction $\bigwedge_\theta \theta(P)$ where θ ranges over all substitutions over the x's. Heuristically, Simplify selects from this infinite conjunction those instances $\theta(P)$ that seem "relevant" to the conjecture (as determined by heuristics described below), asserts the relevant instances, and treats these assertions by the quantifier-free reasoning methods described previously. In this context, the quantified variables $x_1, \ldots x_n$ are called *pattern variables*.

The basic idea of the relevance heuristics is to treat an instance $\theta(P)$ as relevant if it contains enough terms that are represented in the current E-graph. The simplest embodiment of this basic idea is to select a particular term t from P as a *trigger*, and to treat $\theta(P)$ as relevant if $\theta(t)$ is represented in the E-graph. (Simplify also allows a trigger to be a list of terms instead of a single term, as described later in this subsection).

The part of Simplify that finds those substitutions θ such that $\theta(t)$ is represented in the E-graph and asserts the corresponding instance $\theta(P)$ is called the *matcher*, since the trigger plays the role of a pattern that must be matched by some E-node.

The choice of a trigger is heuristically crucial. If too liberal a trigger is chosen, Simplify can be swamped with irrelevant instances; if too conservative a trigger is chosen, an instance crucial to the proof might be excluded. At a minimum, it is important that every one of the pattern variables occur in the trigger, since otherwise there will be infinitely many instances that satisfy the relevance criterion.

As an example of the effect of trigger selection, consider the quantified formula

$$(\forall x, y : car(cons(x, y)) = x).$$

If this is used with the trigger $cons(x, y)$, then, for each term of the form $cons(a, b)$ represented in the E-graph, Simplify will assert $a = car(cons(a, b))$, creating a new E-node labeled *car* if necessary. If instead the formula is used with the more restrictive trigger $car(cons(x, y))$, then the equality will be asserted only when the term $car(cons(a, b))$ is already represented in the E-graph. For the conjecture

$$cons(a, b) = cons(c, d) \Rightarrow a = c,$$

the liberal trigger would allow the proof to go through, while the more conservative trigger would fail to produce the instances necessary to the proof.

One of the pitfalls threatening the user of Simplify is the *matching loop*. For example, an instance of a quantified assertion A might trigger a new instance of a quantified assertion B which in turn triggers a new instance of A, and so on indefinitely.

Simplify has features that try to prevent matching loops from occurring, namely the activation heuristic of Section 5.2, and the trigger selection "loop test" of Section 5.3. However, they don't eliminate matching loops entirely, and Simplify has a feature that attempts to detect when one has occurred, namely the "consecutive matching round limit" of Section 5.2.

Sometimes we must use a set of terms as a trigger instead of a single term. For example, for a formula like

$$(\forall s, t, x : member(x, s) \land subset(s, t) \Rightarrow member(x, t)),$$

no single term is an adequate trigger, since no single term contains all the pattern variables. An appropriate trigger is the set of terms $\{member(x, s), subset(s, t)\}$. A trigger that contains more than one term will be called a *multitrigger*, and the terms will be called its *constituents*. A trigger with a single constituent will be called a *unitrigger*.

Recall from Section 4.2 that when we say that an instance $\theta(t)$ is represented in an E-graph, we mean that it is represented up to congruence. For example, consider the E-graph that represents the equality $f(a) = a$. It has only two E-nodes, but it represents not just a and $f(a)$ but also $f(f(a))$ and indeed $f^n(a)$ for any n.

Matching in the E-graph is more powerful than simple conventional pattern-matching, since the matcher is able to exploit the equality information in the E-graph. For example, consider proving that

$$g(f(g(a))) = a$$

follows from

$$(\forall x : f(x) = x) \tag{6}$$

(for which we assume the trigger $f(x)$) and

$$(\forall x : g(g(x)) = x) \tag{7}$$

(for which we assume the trigger $g(g(x))$). The E-graph representing the query (§2) contains the term $g(f(g(a)))$. A match of (6) with the substitution $x := g(a)$ introduces the equality $f(g(a)) = g(a)$ into the graph. The resulting E-graph is shown in the figure to the right. By virtue of the equality, the resulting E-graph represents an instance of the trigger $g(g(x))$, and the associated instance of (7) (via the substitution $x := a$) completes the proof.

The standard top-down pattern-matching algorithm can be modified slightly to match in an E-graph; the resulting code is straightforward, but because of the need to search each equivalence class, the matcher requires exponential time in the worst case. Indeed, the problem of testing whether an E-node of an E-graph is an instance of a trigger is *NP*-complete, as has been proved by Dexter Kozen [Kozen 1977]. More details of the matching algorithm are presented below.

In practice, although the cost of matching is significant, the extra power derived by exploiting the equalities in the matcher is worth the cost. Also, in our experience, whenever Simplify was swamped by a combinatorial explosion, it was in the backtracking search in *Sat*, not in the matcher.

Although Simplify's exploits the equalities in the E-graph, it does not exploit the laws of arithmetic. For example, Simplify fails to prove

$$(\forall x : P(x + 1)) \Rightarrow P(1 + a)$$

since the trigger $x + 1$ doesn't match the term $1 + a$.

There seem to be two approaches that would fix this.

The first approach would be to write a new matcher that encodes the partial match to be extended by the matching iterators not as a simple binding of pattern variables to equivalence classes but as an affine space of such bindings to be refined by the iterator.

The second approach would be to introduce axioms for the commutativity and associativity of the arithmetic operators so that the E-graph would contain many more ground terms (§2) that could be matched. In the example above, the equivalence class of $P(1 + a)$ would include $P(a + 1)$. This method was used in the Denali superoptimizer [Joshi et al. 2002], which uses Simplify-like techniques to generate provably optimal machine code.

The first approach seems more complete than the second. Presumably, it would find that $P(a)$ is an instance of $P(x + 1)$ by the substitution $x := a - 1$, while the second method, at least as implemented in Denali, is not so aggressive as to introduce the E-node $(a - 1) + 1$ and equate it with the node for a.

But in the course of the ESC project, we never found that Simplify's limitations in using arithmetic information in the matcher were fatal to the ESC application. Also, each approach contains at least a threat of debilitating combinatorial explosion, and neither approach seems guaranteed to find enough matches to substantially increase Simplify's power. So we never implemented either approach.

Simplify transforms quantified formulas into data structures called matching rules. A *matching rule mr* is a triple consisting of a *body*, *mr.body*, which is a formula; a list of variables *mr.vars*; a list of triggers *mr.triggers*, where each trigger is a list of one or more terms. There may be more than one trigger, since it may be heuristically desirable to trigger instances of the quantified formula for more than one ground term; a trigger may have more than one term, since it may be a multi-trigger instead of a uni-trigger.

Simplify maintains a set of "asserted matching rules" as part of its context. Semantically, the assertion of a matching rule *mr* is equivalent to the assertion of ($\forall$ *mr.vars* : *mr.body*). Heuristically, Simplify will use only those instances $\theta(mr.body)$ of the matching rule such that for some trigger *tr* in *mr.triggers*, for each constituent *t* of *tr*, $\theta(t)$ is represented in the E-graph.

There are three more topics in the story of quantifiers and matching rules.

—Matching and backtracking search: how the backtracking search makes use of the set of asserted matching rules,

—Quantifiers to matching rules: how and when quantified formulas are turned into matching rules and matching rules are asserted, and

—How triggers are matched in the E-graph.

These three topics are discussed in the next three sections.

5.2. MATCHING AND BACKTRACKING SEARCH.   In this section, we describe how the presence of asserted matching rules interacts with the backtracking search in *Sat*.

We will make several simplifying assumptions throughout this section:

First, we assume that there is a global set of matching rules fixed for the whole proof. Later, we will explain that the set of asserted matching rules may grow and shrink in the course of the proof, but this doesn't affect the contents of this section in any interesting way.

Second, we assume that the body of every asserted matching rule is a clause, that is, a disjunction of literals. We will return to this assumption in Section 5.3.

Third, we assume that we have an algorithm for enumerating substitutions θ that are relevant to a given trigger *tr*. Such an algorithm will be presented in Section 5.4.1.

The high-level description of the interaction of searching and matching is very simple: periodically during the backtracking search, Simplify performs a "round of matching", in which all relevant instances of asserted matching rules are constructed and added to the clause set, where they become available for the subsequent search. Before we present the detailed description, we make a few high-level points.

First, when Simplify has a choice between matching and case splitting, it favors matching.

Second, Simplify searches for matches of rules only in the portion of the E-graph that represents the literals that have been assumed true or false on the current path (§3.1). This may be a small portion of the E-graph, since there may be many E-nodes representing literals that have been created but not yet been selected for a case split. This heuristic, called the *activation heuristic*, ensures that Simplify will to some extent alternate between case-splitting and matching, and therefore avoids matching loops. Disabling this heuristic has a disastrous effect on performance (see Section 9.8). To implement the heuristic, we maintain an *active* bit in every E-node. When a literal is asserted, the active bit is set in the E-nodes that represent

the literal. More precisely, the bit is set in each E-node equivalent to any subterm of any term that occurs in the literal. All changes to active bits are undone by *Pop*. The policy of matching only in the active portion of the E-graph has one exception, the "select-of-store" tactic described in Section 5.2.1 below.

Third, the first time a clause is created as an instance of a matching rule, we find it worthwhile to refine it aggressively, by plunging (§4.6). If a literal is found to be untenable by plunging, it is deleted from the clause and also explicitly denied (a limited version of the subsumption heuristic (§3.2)). We also use refinement on every clause in the clause set before doing any case split, but in this case we use less aggressive refinement, by status and E-graph tests only.

Fourth, Simplify maintains a set of *fingerprints* of matches that have been found on the current path. To see why, consider the case in which the matcher produces a clause G and then deeper in the proof, in a subsequent round of matching, rediscovers G. It would be undesirable to have two copies of G in the clause set. Therefore, whenever Simplify adds to the clause set an instance $\theta(R.body)$ of a matching rule R by a substitution θ, it also adds the fingerprint of the pair (R, θ) to a set *matchfp*. To filter out redundant instances, this set is checked as each match is discovered. Insertions to *matchfp* are undone by *Pop*.

In general, a fingerprint is like a hash function, but is computed by a CRC algorithm that provably makes collisions extremely unlikely [Rabin 1981]. To fingerprint an instance θ of a matching rule m, we use the CRC of the integer sequence

$$(i, \theta(m.vars[1]).root.id, \ldots \theta(m.vars[m.vars.length]).root.id)$$

where i is the index of the matching rule in the set of matching rules. (The *id* field of an E-node is simply a unique numeric identifier; see Section 7.1.) This approach has the limitation that the root of a relevant equivalence class may have changed between the time a match is entered in the fingerprint table and the time an equivalent match is looked up, leading to the instantiation of matches that are in fact redundant. We don't think such a false miss happens very often, but we don't know for sure. To the extent that it does happen, it reduces the effectiveness the fingerprint test as a performance heuristic, but doesn't lead to any unsoundness or incompleteness.

If a fingerprint collision (false hit) did occur, it could lead to incompleteness, but not unsoundness. We attempted once to switch from 64-bit fingerprints to 32-bit fingerprints, and found that this change caused incompleteness on our test suite, so we retracted the change. We have never observed anything to make us suspect that any collisions have occurred with 64-bit fingerprints.

Here is the place for a few definitions that will be useful later.

First, a redefinition: we previously followed tradition in defining a substitution as a map from variables to terms, but from now on we will treat a substitution as a map from variables to equivalence classes in the E-graph. Thus, $\theta(v)$ is properly an equivalence class and not a term; in contexts where a term is required, we can take any term represented by the equivalence class.

We say that a substitution θ *matches* a term t to an E-node v if (1) the equalities in the E-graph imply that $\theta(t)$ is equal to v, and (2) the domain of θ contains only variables free in t.

We say that a substitution θ *matches* a list $t_1, \ldots, t_n$ of terms to a list $v_1, \ldots, v_n$ of E-nodes if (1) the equalities in the E-graph imply that $\theta(t_i)$ is equal to v_i, for each i, and (2) the domain of θ contains only variables free in at least one of the t's.

We say that a substitution θ *matches* a term t to the E-graph if there exists some active (§5.2) E-node v such that θ matches t to v.

We say that a substitution θ *matches* a list $t_1, \ldots, t_n$ of terms to the E-graph if there exists some list $v_1, \ldots, v_n$ of active E-nodes such that θ matches $t_1, \ldots, t_n$ to $v_1, \ldots, v_n$.

These definitions are crafted so that the set of substitutions that match a trigger to the E-graph is (1) finite and (2) does not contain substitutions that are essentially similar to one another. Limiting the domain of a substitution to variables that appear in the term or term list is essential for (1), and treating substitutions as maps to E-graph equivalence classes instead of terms is essential for both (1) and (2).

Matching is added to the backtracking search from within the procedure *Refine* (§3.2). Recall that the purpose of *Refine* is to perform tactics, such as width reduction and unit assertion, that have higher priority than case splitting. In addition to using a Boolean to keep track of whether refinement is enabled, the new version of this procedure uses a Boolean to keep track of whether matching has any possibility of discovering anything new. Here is the code:

```
proc Refine() ≡
  loop
    if refuted or cls contains an empty clause then exit end;
    if cls contains any unit clauses then
        assert all unit clauses in cls;
        enable refinement;
        enable matching;
    else if refinement enabled then
        for each clause C in cls do
          refine C by cheap tests (possibly setting refuted)
        end;
        disable refinement
    else if matching enabled then
        for each asserted matching rule M do
          for each substitution θ
              that matches some trigger in M.triggers to the E-graph do
              let fp = fingerprint((M, θ)) in
                if not fp ∈ matchfp then
                    add fp to matchfp;
                    let C = θ(M.body) in
                        refine C by plunging (possibly setting refuted);
                        add C to cls
                    end
                end
          end
        end;
        disable matching
    else
        exit
    end
  end
end
```

When *Refine* returns, either the context has become unsatisfiable (in which case *Sat* backtracks) or Simplify's unconditional inference methods have been exhausted, in which case *Sat* performs a case split, as described previously.

There are two additional fine points to mention that are not reflected in the code above.

First, Simplify distinguishes *unit matching rules*, whose bodies are unit clauses, from *non-unit matching rules*, and maintains separate enabling Booleans for the two kinds of rules. The unit rules are matched to quiescence before the nonunit rules are tried at all. In retrospect, we're not sure whether this distinction was worth the trouble.

The second fine point concerns the *consecutive matching round limit*, which is a limit on the number of consecutive rounds of non-unit matching that Simplify will perform without an intervening case split. If the limit is exceeded, Simplify reports a "probable matching loop" and aborts the proof.

5.2.1. *The Matching Depth Heuristic.*   Matching also affects the choice of which case split to perform.

We have mentioned the goal property (§3.5) and the score (§3.6) as criteria for choosing case splits; an even more important criterion is the *matching depth* of a clause. We define by mutual recursion a *depth* for every clause and a *current depth* at any point on any path of the backtracking search: The current depth is initially zero and in general is the maximum depth of any clause that has been split on in the current path. The depth of all clauses of the original query, including defining clauses introduced by proxies (§3.3), is zero. The depth of a clause produced by matching is one greater than the current depth at the time it was introduced by the matcher.

Now we can give the rule for choosing a split: favor low depths; break depth ties by favoring goal clauses; break depth-and-goal ties by favoring high scores.

Implementation note: At any moment, Simplify is considering clauses only of the current depth and the next higher depth. Therefore, Simplify does not store the depth of a clause as part of the clause's representation, but instead simply maintains two sets of clauses, the *current clause set* containing clauses of the current depth, and the *pending clause set* containing clauses of the next higher depth. Only clauses in the current clause set are candidates for case splitting. Clauses produced by matching are added to the pending clause set. When the current clause set becomes empty, Simplify increases the matching depth: the current set gets the pending set, and the pending set gets the empty set.

An additional complexity is that some clauses get *promoted*, which reduces their effective depth by one. Given the two-set implementation described in the previous note, a clause is promoted simply by putting it into the current clause set instead of the pending clause set. Simplify performs promotion for two reasons: *merit promotion* and *immediate promotion*.

Merit promotion promotes a limited number of high scoring clauses. A *promote set* of fingerprints of high-scoring clauses is maintained and used as follows. Whenever *Pop* reduces the matching depth, say from $d + 1$ to d, the clauses of depth $d + 1$ (which are about to be removed by *Pop*) are scanned, and the one with the highest score whose fingerprint is not already in the promote set has its fingerprint added to the promote set. When choosing a case split, Simplify effectively treats all clauses whose fingerprints are in the promote set as if they were in the current clause set, and also increases their effective scores. Insertions to the promote set are not undone by *Pop*, but the promote set is cleared whenever scores are renormalized. Also, there is a bound on the size of the promote set (defaulting to 10 and

settable by an environment variable); when adding a fingerprint to the promote set, Simplify will, if necessary, delete the oldest fingerprint in the set to keep the size of the set within the bound.

Immediate promotion promotes all instances of certain rules that are deemed *a priori* to be important. Simplify's syntax for quantified formulas allows the user to specify that instances are to be added directly to the current clause set rather than to the pending clause set. In our ESC application, the only quantified formula for which immediate promotion is used is the non-unit select-of-store axiom:

$$(\forall a, i, x, j : i = j \lor select(store(a, i, x), j) = select(a, j)).$$

There is a bound (defaulting to 10 and settable by an environment variable) limiting the number of consecutive case splits that Simplify will perform on immediately promoted clauses in preference to other clauses.

The nonunit select-of-store axiom is so important that it isn't surprising that it is appropriate to treat it to immediate promotion. In fact, when working on a challenging problem with ESC/Modula-3, we encountered a proof obligation on which Simplify spent an unacceptable amount of time without succeeding, and analysis revealed that on that problem, even immediate promotion was an insufficiently aggressive policy. The best strategy that we could find to correct the behavior was to add the *select-of-store* tactic, which searches for instances of the trigger $select(store(a, i, x), j))$ even in the inactive portion of the E-graph. For each such instance, the tactic uses the E-graph tests (§4.6) to check whether the current context implies either $i = j$ or $i \neq j$, and if so, the application of select is merged either with x or $select(a, j)$ as appropriate. Because of the memory of this example, Simplify enables the select-of-store tactic by default, although on the test suites described in Section 9 the tactic has negligible performance effects.

In programming Simplify, our policy was to do what was necessary to meet the requirements of the ESC project. One unfortunate consequence of this policy is that the clause promotion logic became overly complicated. Merit promotion and immediate promotion work as described above, and they are effective (as shown by the data in Sections 9.6 and 9.7). But we now report somewhat sheepishly that as we write we cannot find any examples for which the bounds on promote set size and on consecutive splits on immediately promoted clauses are important, although our dim memory is that we originally added those bounds in response to such examples.

In summary, we feel confident that ordering case splits by depth is generally a good idea, but exceptions must sometimes be made. We have obtained satisfactory results by promoting high-scoring clauses and instances of the select-of-store axiom, but a clean, simple rule has eluded us.

5.3. QUANTIFIERS TO MATCHING RULES. In this section, we describe how and when quantified formulas get turned into matching rules.

We begin with a simple story and then describe the gory details.

5.3.1. *Simple Story.* We define a *basic literal* to be a nonproxy literal.

The query (§2) is rewritten as follows: $\Rightarrow$ and $\Leftrightarrow$ are eliminated by using the following equations

$$P \Leftrightarrow Q = (P \Rightarrow Q) \land (Q \Rightarrow P)$$
$$P \Rightarrow Q = (\neg P \lor Q).$$

Also, all occurrences of $\neg$ are driven down to the leaves (i.e., the basic literals), by using the following equations:

$$\neg (P \wedge Q) = (\neg P) \vee (\neg Q)$$
$$\neg (P \vee Q) = (\neg P) \wedge (\neg Q)$$
$$\neg ((\forall x : P)) = (\exists x : \neg P)$$
$$\neg ((\exists x : P)) = (\forall x : \neg P)$$
$$\neg \neg P = P.$$

Then existential quantifiers are eliminated by Skolemizing. That is, we replace each subformula of the form $(\exists y : Q)$ with $Q(y := f(x_1, \ldots, x_n))$, where f is a uniquely named Skolem function and the x's are the universally quantified variables in scope where the subformula appears.

Finally, adjacent universal quantifiers are collapsed, using the rule

$$(\forall x : (\forall y : P)) = (\forall x, y : P).$$

Thus, we rewrite the query into a formula built from $\wedge$, $\vee$, $\forall$, and basic literals. We say that the formula has been put into *positive form*.

The elimination rule for $\Leftrightarrow$ can potentially cause an exponential explosion, but in our application we have not encountered deep nests of $\Leftrightarrow$, and the rule has not been a problem. The other elimination rules do not increase the size of the formula.

Once the formula has been put into positive form, we apply *Sat* to the formula as described previously. This leads to a backtracking search as before, in which basic literals and proxy literals are asserted in an attempt to find a path of assertions that satisfies the formula. What is new is that the search may assert a universally quantified formula in addition to a basic literal or proxy literal. Technically, each universally quantified formula is embedded in a *quantifier proxy* which is a new type of literal that can occur in a clause. Asserting a quantifier proxy causes the universally quantified formula embedded in it to be converted to one or more matching rules, and causes these rules to be asserted.

Thus, it remains only to describe how universally quantified positive formulas are turned into matching rules.

To turn $(\forall x_1, \ldots, x_n : P)$ into matching rules, we first rewrite P into CNF (true CNF, not the equisatisfiable CNF (§3.3) used in *Sat*). The reason for this is that clausal rules are desirable, since if the body of a rule is a clause, we can apply width reduction and clause elimination (§3.2) to instances of the rule. By rewriting P into a conjunction of clauses, we can distribute the universal quantifier into the conjunction and produce one clausal rule for each clause in the CNF for P. For example, for the quantified formula

$$(\forall x : P(x) \Rightarrow (Q(x) \wedge R(x))),$$

we rewrite the body as a conjunction of two clauses and distribute the quantifier into the conjunction to produce two clausal rules, as though the input had been

$$(\forall x : P(x) \Rightarrow Q(x)) \wedge (\forall x : P(x) \Rightarrow R(x)).$$

Then, for each rule, we must choose one or more triggers.

Simplify's syntax for a universal quantifier allows the user to supply an explicit list of triggers. If the user does not supply an explicit list of triggers, then Simplify makes two tries to select triggers automatically.

First try: Simplify makes a unitrigger (§5.1) out of any term that (1) occurs in the body outside the scope of any nested quantifier, (2) contains all the quantified variables, (3) passes the "loop test", (4) is not a single variable, (5) is not "proscribed", and (6) contains no proper subterm with properties (1)–(5).

The loop test is designed to avoid infinite loops in which a matching rule creates larger and larger instances of itself—that is, matching loops involving a single rule. A term fails the loop test if the body of the quantifier contains a larger instance of the term. For example, in

$$(\forall x : P(f(x), f(g(x)))),$$

the term $f(x)$ fails the loop test, since the larger term $f(g(x))$ is an instance of $f(x)$ via $x := g(x)$. Thus, in this case, $f(x)$ will not be chosen as a trigger, and Simplify will avoid looping forever substituting $x := t$, $x := g(t)$, $x := g(g(t))$, ...

The proscription condition (4) is designed to allow the user to exclude certain undesirable triggers that might otherwise be selected automatically. Simplify's input syntax for a universal quantifier also allows the user to supply an explicit list of terms that are *not* to be used as triggers; these are the terms that are "proscribed".

Second try: If the first try doesn't produce any unitriggers, then Simplify tries to create a multitrigger (§5.1). It attempts to select as the constituents (§5.1) a reasonably small set of nonproscribed terms that occur in the body outside the scope of any nested quantifier and that collectively contain all the pattern variables and (if possible) overlap somewhat in the pattern variables that they contain. If the second try succeeds, it produces a single multitrigger. Because there could be exponentially many plausible multitriggers, Simplify selects just one of them.

The ESC/Modula-3 and ESC/Java annotation languages allow users to include quantifiers in annotations since the expressiveness provided by quantifiers is occasionally required. But these tools don't allow users to supply triggers for the quantifiers, since the ESC philosophy is to aim for highly automated checking. However, the background predicates (§2) introduced by the ESC tools include many explicit triggers, and these triggers are essential. In summary, for simple quantified assertions (like "all elements of array A are non-null" or "all allocated readers have non-negative count fields") automatic trigger selection seem to work adequately. But when quantifiers are used in any but the simplest ways, Simplify's explicit trigger mechanism is required.

5.3.2. *Gory Details.* A disadvantage of the simple story is that the work of converting a particular quantified formula into a matching rule will be repeated many times if there are many paths in the search that assert that formula. To avoid this disadvantage, we could be more eager about converting quantified formulas into matching rules; for example, we could convert every quantified formula into a matching rule as part of the original work of putting the query into positive form. But this maximally eager approach has disadvantages as well; for example, it may do the work of converting a quantified formula into a matching rule even if the backtracking search finds a satisfying assignment (§2) without ever asserting the formula at all. Therefore, Simplify takes an intermediate approach, not totally eager but not totally lazy, either.

When the query is put into positive form, each outermost quantifier is converted into one or more matching rules. These matching rules are embedded in the quantifier proxy, instead of the quantified formula itself. Thus, the work of building

matching rules for outermost quantifiers is performed eagerly, before the back-tracking search begins. However, universal quantifiers that are not outermost, but are nested within other universal quantifiers, are simply treated as uninterpreted literals: their bodies are not put into positive form and no quantifier proxies are created for them.

When a matching rule corresponding to an outer universal quantifier is instantiated, its instantiated body is asserted, and at this point the universal quantifiers that are outermost in the body are converted into quantifier proxies and matching rules.

Let us say that a formula is in *positive seminormal form* if the parts of it outside universal quantifiers are in positive normal form. Then, in general, the bodies of all matching rules are in positive seminormal form, and when such a body is instantiated and asserted, the outermost quantifiers within it are turned into matching rules (which work includes the work of putting their bodies into positive seminormal form).

That was the first gory detail. We continue to believe that the best approach to the work of converting quantifiers into matching rules is somewhere intermediate between the maximally eager and maximally lazy approaches, but we don't have confidence that our particular compromise is best. We report this gory detail for completeness rather than as a recommendation.

Next we must describe *nonclausal rules*, that is, matching rules whose bodies are formulas other than clauses. In the simple story, the bodies of quantified formulas were converted to true CNF, which ensured that all rule bodies would be clauses. Clausal rules are desirable, but conversion to true CNF can cause an exponential size explosion, so we need an escape hatch. Before constructing the true CNF for the body of a quantified formula, Simplify estimates the size of the result. If the estimate is large, it produces a matching rule whose body is the unnormalized quantifier body. The trigger for such a rule is computed from the atomic formulas (§2) in the body just as if these atomic formulas were the elements of a clause. When a nonclausal rule is instantiated, the instantiation of its body is asserted and treated by the equisatisfiable CNF methods described previously.

Recall that Simplify favors case splits with low matching depth (§5.2.1). This heuristic prevents Simplify from fruitlessly searching all cases of the latest instance of a rule before it has finished the cases of much older clauses. We have described the heuristic for clausal rules. Nonclausal rules introduce some complexities; for example, the equisatisfiable CNF for the instance of the rule body may formally be a unit clause, consisting of a single proxy; Simplify must not let itself be tricked into asserting this proxy prematurely.

Finally, Simplify's policy of treating nested quantifiers as uninterpreted literals can cause trigger selection to fail. For example, in the formula

$$(\forall x, y : P(x) \implies (\forall z : Q(x, y) \wedge Q(z, x)))$$

both tries to construct a trigger will fail. Instead of failing, it might be better to in such a case to move $\forall z$ outwards or $\forall y$ inwards. But we haven't found trigger selection failure to be a problem in our applications. (Of course, if we didn't collapse adjacent universal quantifiers, this problem would arise all the time.)

In the case of nested quantifiers, the variables bound by the outer quantifier may appear within the inner quantifier. In this case, when the matching rule for the outer quantifier is instantiated, an appropriate substitution must be performed on

the matching rule corresponding to the inner quantifier. For example, consider

$$(\forall x : P(x) \;\Rightarrow\; (\forall y : Q(x, y))).$$

If the outer rule is instantiated with $x := E$, then the substitution $x := E$ is performed on the body of the outer rule, which includes the inner matching rule. Thus, the body of the inner rule will be changed from $Q(x, y)$ to $Q(E, y)$.

Our view that nested quantifiers should produce nested matching rules should be contrasted with the traditional approach of putting formulas into prenex form by moving the quantifiers to the outermost level. Our approach is less pure, but it allows for more heuristic control. For example, the matching rule produced by asserting the proxy for

$$(\forall x : \neg P(x) \vee (\forall y : Q(x, y))) \tag{8}$$

is rather different from the matching rule produced by asserting the proxy for the semantically equivalent

$$(\forall x, y : \neg P(x) \vee Q(x, y)). \tag{9}$$

In the case of (8), the matching rule will have trigger $P(x)$, and if it is instantiated and asserted with the substitution $x := E$, the clause $\neg P(E) \vee (\forall y : Q(E, y))$ will be asserted. If the case analysis comes to assert the second disjunct, the effect will be to create and assert a matching rule for the inner quantifier.

The semantically equivalent formula (9) affects the proof very differently: a single matching rule with trigger $Q(x, y)$ would be produced. It is a little disconcerting when semantically equivalent formulas produce different behaviors of Simplify; on the other hand, it is important that the input language be expressive enough to direct Simplify towards heuristically desirable search strategies.

Two final details: First, some transformations are performed as the quantified symbolic expression is transformed into matching rules. One straightforward transformation is the elimination of unused quantified variables. For example, if x does not occur free in P, then $(\forall x, y : P)$ is transformed into $(\forall y : P)$. Less straightforward are the *one-point rules*, of which there are several. The simplest is the replacement of $(\forall x : x \neq T \vee P(x))$ by $P(x := T)$. The one-point rules were important to an early version of ESC/Modula-3, whose treatment of data abstraction produce fodder for the rules. They are also useful for proving trivialities like $(\exists x : x = 3)$. But neither the later versions of ESC/Modula-3 nor ESC/Java seem to exercise the one-point rules. Second, some of the transformations described in this subsection may be reenabled by others. For example, distributing $\forall$ over $\wedge$ may create new opportunities for merging adjacent universal quantifiers or eliminating unused quantified variables. Simplify continues performing these transformations until none is applicable.

5.4. How Triggers Are Matched. Recall that to use the asserted matching rules, the backtracking search performs an enumeration with the following structure:

```
for each asserted matching rule M do
    for each substitution θ
        that matches (§5.2) some trigger in M.triggers to the E-graph do
        ...
    end
end
```

In this section, we will describe how the substitutions are enumerated.

5.4.1. *Matching Iterators.* We present the matching algorithms as mutually recursive iterators in the style of CLU [Liskov et al. 1981]. Each of the iterators takes as arguments one or more terms together with a substitution and yields all ways of extending the substitution that match the term(s). They differ in whether a term or term-list is to be matched, and in whether the match is to be to anywhere in the E-graph or to a specific E-node or list of E-nodes. When the matches are to arbitrary E-nodes, the terms are required to be proper (i.e., not to be single variables).

We say that two substitutions θ and ϕ *conflict* if they map some variable to different equivalence classes.

Here are the specifications of the four iterators:

> **iterator** *MatchTrigger*(t : list of proper terms, θ : substitution)
>   yields all extensions $\theta \cup \phi$ of θ such that
> $\theta \cup \phi$ matches (§5.2) the trigger t in the E-graph and
> ϕ does not conflict with θ.

> **iterator** *MatchTerm*(t : proper term, θ : substitution)
>   yields all extensions $\theta \cup \phi$ of θ such that
> $\theta \cup \phi$ matches t to some active E-node in the E-graph and
> ϕ does not conflict with θ.

> **iterator** *Match*(t : term, v : *E-node*, θ : substitution)
>   yields all extensions $\theta \cup \phi$ of θ such that
> $\theta \cup \phi$ matches t to v, and
> ϕ does not conflict with θ.

> **iterator** *MatchList*(t : list of terms, v : list of *E-node*'s, θ : substitution)
>   yields all extensions $\theta \cup \phi$ of θ such that
> $\theta \cup \phi$ matches the term-list t to the E-node-list v, and
> ϕ does not conflict with θ.

Given these iterators, a round of matching is implemented approximately as follows:

```
for each asserted matching rule M do
  for each trigger tr in M.triggers do
    for each θ in MatchTrigger(tr, { }) do
      ...
    end
  end
end
```

where { } denotes the empty substitution.

We now describe the implementation of the iterators.

The implementation of *MatchTrigger* is a straightforward recursion on the list:

```
iterator MatchTrigger(t : list of proper terms, θ : substitution) ≡
  if t is empty then
    yield(θ)
  else
    for each φ in MatchTerm(hd(t), θ) do
      for each ψ in MatchTrigger(tl(t), φ) do
```

```
              yield(ψ)
            end
          end
        end
    end
```

(We use the operators "hd" and "tl" on lists: hd(l) is the first element of the list l, and tl(l) is the list of remaining elements.)

The implementation of *MatchTerm* searches those E-nodes in the E-graph with the right label, testing each by calling *MatchList*:

```
iterator MatchTerm(t : proper term, θ : substitution) ≡
    let f, args be such that t = f(args) in
        for each active E-node v labeled f do
            for each φ in MatchList(args, children[v], θ) do
                yield(φ)
            end
        end
    end
end
```

The iterator *MatchList* matches a list of terms to a list of E-nodes by first finding all substitutions that match the first term to the first E-node, and then extending each such substitution in all possible ways that match the remaining terms to the remaining E-nodes. The base case of this recursion is the empty list, which requires no extension to the substitution; the other case relies on *Match* to find the substitutions that match the first term to the first E-node:

```
iterator MatchList(t : list of terms, v : list of E-node's, θ : substitution) ≡
    if t is the empty list then
        yield(θ)
    else
        for each φ in Match(hd(t), hd(v), θ) do
            for each ψ in MatchList(tl(t), tl(v), φ) do
                yield(ψ)
            end
        end
    end
end
```

The last iterator to be implemented is *Match*, which finds all ways of matching a single term to a single E-node. It uses recursion on the structure of the term. The base case is that the term consists of a single pattern variable. In this case there are three possibilities: either the substitution needs to be extended to bind the pattern variable appropriately, or the substitution already binds the pattern variable compatibly, or the substitution already contains a conflicting binding for the pattern variable. If the base case does not apply, the term is an application of a function symbol to a list of smaller terms. (We assume that constant symbols are represented as applications of function symbols to empty argument lists, so constant symbols don't occur explicitly as a base case.) To match a proper term to an E-node, we must enumerate the equivalence class of the E-node, finding all E-nodes that are in the desired equivalence class and that have the desired label. For each such E-node, we use *MatchList* to find all substitutions that match the argument terms to the list

of E-node arguments:

```
iterator Match(t : term, v : E-node, θ : substitution) ≡
  if t is a pattern variable then
    if t is not in the domain of θ then
      yield(θ ∪ {(t, v's equivalence class)})
    else if θ(t) contains v then
      yield(θ)
    else
      skip
    end
  else
    let f, args be such that t = f(args) in
      for each E-node u such that u is equivalent to v and
        f is the label of u do
        for each φ in MatchList(args, children[u], θ) do yield(φ) end
      end
    end
  end
end
```

We conclude this section with a few additional comments and details.

There are two places where E-nodes are enumerated: in *Match*, when enumerating those E-nodes *u* that have the desired label and equivalence class, and in *MatchTerm*, when enumerating candidate E-nodes that have the desired label. In both cases, it is important to enumerate only one E-node from each congruence class (§4.2), since congruent E-nodes will produce the same matches. Section 7 shows how to do this.

In *Match*, when enumerating those E-nodes *u* that have the desired label and equivalence class, the E-graph data structure allows two possibilities: enumerating the E-nodes with the desired label and testing each for membership in the desired equivalence class, or vice-versa. A third possibility is to choose between these two based on the whether the equivalence class is larger or smaller than the set of E-nodes with the desired label. Our experiments have not found significant performance differences between these three alternatives.

Simplify uses an optimization that is not reflected in the pseudo-code written above: In *MatchTerm*(t, θ), it may be that θ binds all the pattern variables occurring in *t*. In this case, *MatchTerm* simply checks whether an E-node exists for θ(t), yields θ if it does, and yields nothing if it doesn't.

When Simplify constructs a trigger constituent from an S-expression, subterms that contain no quantified variables (or whose quantified variables are all bound by quantifiers at outer levels) are generally interned (§4.5) into E-nodes at trigger creation time. This produces an extra base case in the iterator *Match*: if *t* is an E-node *w*, then *Match* yields θ if *w* is equivalent to *v*, and yields nothing otherwise.

5.4.2. *The Mod-Time Matching Optimization.*   Simplify spends much of its time matching triggers in the E-graph. The general nature of this matching process was described above. In this section and the next, we describe two important optimizations that speed up matching: the *mod-time* optimization and the *pattern-element* optimization.

To describe the mod-time optimization, we temporarily ignore multitriggers.

Roughly speaking, a round of matching performs the following computation:

```
for each matching rule with uni-trigger T and body B do
  for each active E-node V do
    for each substitution θ such that θ(T) is equivalent to V do
      Assert(θ(B))
    end
  end
end
```

Consider two rounds of matching that happen on the same path in the search tree. We find that in this case, it often happens that for many pairs (T, V), no assertions performed between the two rounds changed the E-graph in any way that affects the set of instances of trigger T equivalent to E-node V, and consequently the set of substitutions that are discovered on the first round of matching is identical to the set discovered on the second round. In this case, the work performed in the second round for (T, V) is pointless, since any instances that it could find and assert have already been found and asserted in the earlier round.

The *mod-time* optimization is the way we avoid repeating the pointless work. The basic idea is to introduce a global counter *gmt* that records the number of rounds of matching that have occurred on the current path (§3.1). It is incremented after each round of matching, and saved and restored by *Push* and *Pop*. We also introduce a field $E.mt$ for each active E-node E, which records the value of *gmt* the last time any proper descendant of E was involved in a merge in the E-graph. The idea is to maintain the "mod-time invariant", which is

```
for all matching rules mr with trigger T and body B,
  for all substitutions θ such that θ(T) is represented in the E-graph,
    either θ(B) has already been asserted, or
    the E-node V that represents θ(T) satisfies V.mt = gmt
```

Given the mod-time invariant, the rough version of a matching round becomes:

```
for each matching rule with trigger T and body B do
  for each active E-node V such that v.mt = gmt do
    for each substitution θ such that θ(T) is equivalent to V do
      Assert(θ(B))
    end
  end
end;
gmt := gmt + 1
```

The reason that the code can ignore those E-nodes V for which $V.mt \neq gmt$ is that the invariant implies that such an E-node matches a rule's trigger only if the corresponding instance of the rule's body has already been asserted. The reason that $gmt := gmt + 1$ maintains the invariant is that after a matching round, all instances of rules that match have been asserted.

The other place where we have to take care to maintain the invariant is when E-nodes are merged. When two E-nodes V and W are merged, we must enumerate all E-nodes U such that the merge might change the set of terms congruent to U in the graph. We do this by calling *UpdateMT(V)* (or, equivalently, *UpdateMT(W)*)

immediately after the merge, where this procedure is defined as follows:

```
proc UpdateMT(V : E-node) ≡
  for each P such that P is a parent (§4.2) of some E-node equivalent to V do
    if P.mt < gmt then
      Push onto the undo stack the triple ("UndoUpdatemt", P, P.mt);
      P.mt := gmt;
      UpdateMT(P)
    end
  end
end
```

In the case of circular E-graphs, the recursion is terminated because the second time an E-node is visited, its mod-time will already have been updated.

So much for the basic idea of the mod-time optimization. We have three details to add to the description.

First, we need to handle the case of rules with multitriggers. This is an awkward problem. Consider the case of a multitrigger $p_1, \ldots, p_n$, and suppose that merges change the E-graph so that there is a new instance θ of the multitrigger; that is, so that each $\theta(p_i)$ is represented in the new E-graph, but for at least one i, the term $\theta(p_i)$ was not represented in the old E-graph. Then, for some i, the E-node representing $\theta(p_i)$ has its mod-time equal to gmt, but this need not hold for all i. Hence, when the matcher searches for an instance of the multitrigger by extending all matches of p_1 in ways that match the other p's, it cannot confine its search to new matches of p_1. Therefore, the matcher considers each multi-trigger $p_1, \ldots, p_n$ n times, using each constituent p_i in turn as a *gating term*. The gating term is matched first, against all E-nodes whose mod-times are equal to gmt, and any matches found are extended to cover the other constituent in all possible ways. In searching for extensions, the mod-times are ignored.

Second, it is possible for matching rules to be created dynamically during the course of a proof. If we tried to maintain the mod-time invariant as stated above when a new rule was created, we would have to match the new rule immediately or reset the *mt* fields of all active E-nodes, neither of which is attractive. Therefore, we give each matching rule a Boolean field *new* which is **true** for matching rules created since the last round of matching on the current path, and we weaken the mod-time invariant by limiting its outer quantification to apply only to matching rules that are not new. During a round of matching, new matching rules are matched against all active E-nodes regardless of their mod-times.

Third, as we mentioned earlier, the matcher restricts its search to the "active" portion of the E-graph. We therefore must set $V.mt := gmt$ whenever an E-node V is activated, so that the heuristic doesn't lead the matcher to miss newly activated instances of the trigger.

Combining these three details, we find that the detailed version of the mod-time optimization is as follows. The invariant is:

```
for all matching rules mr with trigger T₁, . . . , Tₙ and body B
    either mr.new or
    for all substitutions θ such that each θ(Tᵢ) is represented
            by an active E-node in the E-graph,
                either θ(B) has already been asserted, or
                there exists an i such that the E-node V that represents
                θ(Tᵢ) satisfies V.mt = gmt.
```

The algorithm for a round of matching that makes use of this invariant is:

```
for each matching rule mr do
    if mr.new then
        for each tr in mr.triggers do
            for each θ such that (∀t ∈ tr : θ(t) exists and is active) do
                Assert(θ(mr.body))
            end
        end;
        mr.new := false
    else
        for each tr in mr.triggers do
            for each t in tr do
                for each active E-node V such that v.mt = gmt do
                    for each ϕ such that ϕ(t) is equivalent to V do
                        for each extension θ of ϕ such that
                                (∀q ∈ tr : θ(q) exists and is active) do
                            Assert(θ(mr.body))
                        end
                    end
                end
            end
        end
    end;
    gmt := gmt + 1
```

To maintain the invariant, we call $UpdateMT(V)$ immediately after an equivalence class V is enlarged by a merge, we set $V.mt := gmt$ whenever an E-node V is activated, and we set $mr.new :=$ **true** whenever a matching rule mr is created.

More details:

Although we have not shown this in our pseudo-code, the upward recursion in *UpdateMT* can ignore inactive E-nodes.

For any function symbol f, the E-graph data structure maintains a linked list of all E-nodes that represent applications of f. The matcher traverses this list when attempting to match a trigger term that is an application of f. Because of the mod-time optimization, only those list elements with mod-times equal to gmt are of interest. We found that it was a significant time improvement to keep this list sorted in order of decreasing mod-times. This is easily done by moving an E-node to the front of its list when its mod-time is updated. In this case, the record pushed onto the undo stack must include the predecessor of the node whose mod-time is changing.

5.4.3. *The Pattern-Element Matching Optimization.* The mod-time optimization speeds matching by reducing the number of E-nodes examined. The pattern-elements optimization speeds matching by reducing the number of triggers considered.

Consider again two rounds of matching that happen on the same path in the search tree. We find that in this case, it often happens that for many triggers T, no assertions performed between the two rounds changed the E-graph in any way that affects the set of instances of T present in the E-graph. In this case, the work performed in the second round for the trigger T is pointless, since any instances that it finds have already been found in the earlier round.

The *pattern-element* optimization is a way of avoiding the pointless work. The basic idea is to detect the situation that a modification to the E-graph is not relevant

to a trigger, in the sense that the modification cannot possibly have caused there to be any new instances of the trigger in the E-graph. A round of matching need consider only those triggers that are relevant to at least one of the modifications to the E-graph that have occurred since the previous round of matching on the current path.

There are two kinds of modifications to be considered: merges of equivalence classes and activations of E-nodes. To begin with, we will consider merges only. There are two ways that a merge can be relevant to a trigger. To explain these ways, we begin with two definitions.

A pair of function symbols (f, g) is a *parent–child* element of a trigger if the trigger contains a term of the form

$$f(\ldots, g(\ldots), \ldots),$$

that is, if somewhere in the trigger, an application of g occurs as an argument of f.

A pair of (not necessarily distinct) function symbols (f, g) is a *parent–parent* element of a trigger for the pattern variable x if the trigger contains two distinct occurrences of the pattern variable x, one of which is in a term of the form

$$f(\ldots, x, \ldots),$$

and the other of which is in a term of the form

$$g(\ldots, x, \ldots),$$

that is, if somewhere in the trigger, f and g are applied to distinct occurrences of the same pattern variable.

For example, (f, f) is a parent–parent element for x of the trigger $f(x, x)$, but not of the trigger $f(x)$.

The first case in which a merge is relevant to a trigger is "parent–child" relevance, in which, for some parent–child element (f, g) of the trigger, the merge makes some active application of g equivalent to some active argument of f.

The second case is "parent–parent" relevance, in which for some parent–parent element (f, g) of the trigger, the merge makes some active argument of f equivalent to some active argument of g.

We claim that a merge that is not relevant to a trigger in one of these two ways cannot introduce into the E-graph any new instances of the trigger. We leave it to the reader to persuade himself of this claim.

This claim justifies the pattern-element optimization. The basic idea is to maintain two global variables, *gpc* and *gpp*, both of which contain sets of pairs of function symbols. The invariant satisfied by these sets is:

> **for all** matching rules *mr* with trigger P and body B
> > **for all** substitutions θ such that $\theta(P)$ is represented in the E-graph
> > > $\theta(B)$ has already been asserted, or
> > > *gpc* contains some parent-child element of P, or
> > > *gpp* contains some parent-parent element of P.

To maintain this invariant, we add pairs to *gpc* and/or to *gpp* whenever a merge is performed in the E-graph.

To take advantage of the invariant, a round of matching simply ignores those matching rules whose triggers' pattern elements have no overlap with *gpc* or *gpp*. After a round of matching, *gpc* and *gpp* are emptied.

In addition to merges, the E-graph changes when E-nodes are activated. The rules for maintaining *gpc* and *gpp* when E-nodes are activated are as follows: when activating an E-node V labeled f, Simplify

(1) adds a pair (f, g) to *gpc* for each active argument of V that is an application of g,

(2) adds a pair (h, f) to *gpc* for each active E-node labeled h that has V as one of its arguments, and

(3) adds a pair (f, k) to *gpp* for each k that labels an active E-node which has any arguments in common with the arguments of V.

Activation introduces a new complication: activating an E-node can create a new instance of a trigger, even though it adds none of the trigger's parent–parent or parent–child elements to *gpp* or *gpc*. In particular, this can happen in the rather trivial case that the trigger is of the form $f(x_1, \ldots, x_n)$ where the x's are distinct pattern variables, and the E-node activated is an application of f. (In case of a multitrigger, any constituent of this form can suffice, if the pattern variables that are arguments to f don't occur elsewhere in the trigger). To take care of this problem, we define a third kind of pattern element and introduce a third global set.

A function symbol f is a *trivial parent* element of a trigger if the trigger consists of an application of f to distinct pattern variables, or the trigger is a multitrigger one of whose constituents is an application of f to distinct pattern variables that do not occur elsewhere in the multitrigger.

We add another global set *gp* and add a fourth alternative to the disjunction in our invariant:

> either $\theta(B)$ has already been asserted, or
> *gpc* contains some parent-child element of P, or
> *gpp* contains some parent-parent element of P, or
> *gp* contains some trivial parent element of P.

The related changes to the program are: we add f to *gp* when activating an application of f, we empty *gp* after every round of matching, and a round of matching does not ignore a rule if the rule's trigger has any trivial parent element in common with *gp*.

5.4.4. *Implementation of the Pattern-Element Optimization.* We must implement the pattern-element optimization carefully if it is to repay more than it costs.

Since exact set operations are expensive, we use *approximate sets*, which are like real sets except that membership and overlap tests can return false positives.

First, we consider sets of function symbols (like *gp*). We fix a hash function *hash* whose domain is the set of function symbols and whose range is [0..63] (for our sixty-four bit machines). The idea is that the true set S of function symbols is represented approximately by a word that has bit $hash(s)$ set for each $s \in S$, and its other bits zero. To test if u is a member of the set, we check if bit $hash(u)$ is set in the word; to test if two sets overlap, we test if the bitwise AND of the bit vectors is non-zero.

Next we consider sets of pairs of function symbols (like *gpc* and *gpp*). In this case, we use an array of 64 words; for each (f, g) in the set, we set bit $hash(g)$ of word $hash(f)$. When adding an unordered pair (f, g) to *gpp*, we add either (f, g) or (g, f) to the array representation, but not necessarily both. At the beginning of

a round of matching, when *gpp* is about to be read instead of written, we replace it with its symmetric closure (its union with its transpose).

For sparse approximate pair sets, like the set of parent–parent or parent–child elements of a particular matching rule, we compress away the empty rows in the array, but we don't do this for *gpp* or *gpc*.

With each equivalence class root *q* in the E-graph, we associate two approximate sets of function symbols: *q.lbls* and *q.plbls*, where *q.lbls* is the set of labels of active E-nodes in *q*'s class and *q.plbls* is the set of labels of active parents of E-nodes in *Q*'s class. The code for merging two equivalence classes, say the class with root *r* into the class with root *q*, is extended with the following:

```
proc UpdatePatElems(q, r : E-node) ≡
    gpc := gpc ∪ r.plbls × q.lbls ∪ r.lbls × q.plbls;
    gpp := gpp ∪ r.plbls × q.plbls;
    Push onto the undo stack the triple ("UndoLabels", q, q.lbls);
    q.lbls := q.lbls ∪ r.lbls;
    Push onto the undo stack the triple ("UndoPLabels", q, q.plbls);
    q.plbls := q.plbls ∪ r.plbls
end
```

The operations on *r.lbls* and *r.plbls* must be undone by *Pop*. This requires saving the old values of the approximate sets on the undo stack (§3.1). Since Simplify does a case split only when matching has reached quiescence, *gpc* and *gpp* are always empty when *Push* is called. Thus, *Pop* can simply set them to be empty, and there is no need to write an undo record when they are updated.

We also must maintain *lbls*, *plbls*, *gpc*, *gpp*, and *gp* when an E-node is activated, but we omit the details.

In addition to function symbols and pattern variables, a trigger can contain E-nodes as a representation for a specific ground term (§2). Such E-nodes are called *trigger-relevant*. For example, consider the following formula:

$$(\forall x : P(f(x)) \Rightarrow (\forall y : g(y, x) = 0)).$$

The outer quantification will by default have the trigger $f(x)$. When it is matched, let V be the E-node that matches x. Then a matching rule will be created and asserted that represents the inner quantification. This rule will have the trigger $g(y, V)$, which contains the pattern variable y, the function symbol g, and the constant trigger-relevant E-node V. For the purpose of computing *gpc* and the set of parent–child elements of a trigger, we treat each trigger-relevant E-node as though it were a nullary function symbol. For example, after the matching rule corresponding to the inner quantification is asserted, then a merge of an argument of g with an equivalence class containing V would add the parent–child element (g, V) to *gpc*. Trigger-relevant E-nodes also come up without nested quantifiers, if a trigger contains ground terms (terms with no occurrences of pattern variables), such as NIL or Succ(0).

5.4.5. *Mod-Times and Pattern Elements and Multitriggers.*   There is one final optimization to be described that uses pattern elements and mod-times. Recall from Section 5.4.2 that, when matching a multitrigger, each constituent of the multitrigger is treated as a gating term (to be matched only against E-nodes with the current mod-time; each match of the gating term is extended to match the other constituents of the multitrigger, without regard to mod-times). The need to consider each constituent

of the multitrigger as a gating term is expensive. As a further improvement, we will show how to use the information in the global pattern-element sets to reduce the number of gating terms that need to be considered when matching a multitrigger.

As a simple example, consider the multitrigger

$$f(g(x)), h(x).$$

An E-graph contains an instance of this multitrigger if and only if there exist E-nodes U, V, and W (not necessarily distinct) satisfying the following conditions:

(1) $f(U)$ is active,
(2) $g(V)$ is active,
(3) $h(W)$ is active,
(4) U is equivalent to $g(V)$, and
(5) V is equivalent to W.

Now suppose that, when this multitrigger is considered for matching, $gpc = \{(f, g)\}$ and gpp is empty. For any new instance (U, V, W) of the multitrigger, one of the five conditions must have become true last. It cannot have been condition (2), (3), or (5) that became true last, since any modification to the E-graph that makes (2), (3), or (5) become the last of the five to be true also introduces (g, h) into gpp. Consequently, for each new instance of the multitrigger, the modification to the E-graph that made it become an instance (the *enabling modification*) must have made condition (1) or (4) become the last of the five to be true. However, any modification to the E-graph that makes (1) or (4) become the last of the five to be true also updates the mod-time of the E-node $f(U)$. Consequently, in this situation, it suffices to use only $f(g(x))$ and not $h(x)$ as a gating term.

As a second example, consider the same multitrigger, and suppose that when the multitrigger is considered for matching, gpc and gp are empty and $gpp = \{(g, h)\}$. In this case, we observe that the last of the five conditions to be satisfied for any new instance (U, V, W) of the multitrigger cannot have been (1), (2), or (4), since satisfying (1), (2), or (4) last would add (f, g) to gpc; nor can it have been (3) that was satisfied last, since satisfying (3) last would add h to gp. Therefore, the enabling modification must have satisfied condition (5), and must therefore have updated the mod-time of both $f(U)$ and $g(V)$. Consequently, in this situation, we can use either $f(g(x))$ or $h(x)$ as a gating term; there is no reason to use both of them.

As a third example, consider the same multitrigger, and suppose that when the multitrigger is considered for matching, $gpc = \{(f, g)\}$, $gpp = \{(g, h)\}$, and gp is empty. In this case, the enabling modification must either have satisfied condition 4 last, updating the mod-time of $f(U)$, or have satisfied condition (5) last, updating the mod-times of both $f(U)$ and $h(W)$. Consequently, in this situation, it suffices to use only $f(g(x))$ as the gating term; it would not suffice to use only $h(x)$.

In general, we claim that when matching a multitrigger $p_1, \ldots, p_n$, it suffices to choose a set of gating terms such that

(1) Each p_i is included as a gating term if it contains any parent–child element in gpc;
(2) For each x_i, g, h such that (g, h) is in gpp and the trigger has (g, h) as a parent–parent element for x_i, the set of gating terms includes some p_j that contains pattern variable x_i; and

(3) Each p_i is included as a gating term if it has the form $f(v_1, \ldots, v_k)$ where the v's are pattern variables, f is in *gp*, and for each v_j, either the trigger has no parent–parent elements for v_j or some parent–parent element for v_j is in *gpp*.

We now justify the sufficiency of the three conditions.

We begin with some definitions. We define a term to be of *unit depth* if it has no parent–child elements. We define a multitrigger to be *tree-like* if none of its pattern variables occurs more than once within it. We define the *shape* of a multitrigger T as the tree-like multitrigger obtained from T be replacing the ith occurrence of each pattern variable x with a uniquely inflected form x_i. For example, the shape of $f(x, x)$ is $f(x_1, x_2)$ Matching a multitrigger is equivalent to matching its shape, subject to the side conditions that all inflections of each original pattern variable are matched to the same equivalence class. The enabling modification to the E-graph that creates a match θ to a multitrigger must either have created a match of its shape or satisfied one of its side conditions. We consider these two cases in turn.

If the enabling modification created a match of the shape of the multitrigger, then it must have created a match of some constituent (§5.1) of that trigger. There are two subcases. If the constituent is not of unit depth, then the enabling modification must have added to *gpc* a parent–child element of some constituent of the multitrigger, and updated the mod-time of the E-node matched by that constituent. By condition (1), therefore, the instance will not be missed. If the constituent is of unit depth, then the enabling modification must be the activation of the E-node matched by the constituent. In this case, the enabling modification must have added the constituent's function symbol *gp* and, for each pattern variable x that occurs in the constituent and that occurs more than once in the entire multitrigger, it must have added some parent–parent element for x to *gpp*. By condition (3), therefore, the instance will not be missed.

If the enabling modification created a match by satisfying one of the side conditions, then it must have added to *gpp* a parent–parent element for some pattern variable, and updated the mod-times of the all the E-nodes matched by constituents containing that variable. By condition (2), therefore, the instance will not be missed.

To construct a set of gating terms, Simplify begins with the minimum set that satisfies conditions (1) and (3), and then for each g, h, x_i as required by condition (2), the set of gating terms is expanded if necessary. The final result will depend on the order in which the different instances of condition (2) are considered, and may not be of minimum size, but in practice this approach reduces the number of gating terms enough to more than pay for its cost on average.

Even with the mod-time and pattern-element optimizations, many of the matches found by the matcher are redundant. Thus, we are still spending some time discovering redundant matches. But in all or nearly all cases, the fingerprint (§5.2) test detects the redundancy, so the unnecessary clauses are not added to the clause set.

6. *Reporting Errors*

Many conjectures are not theorems. Simplify's backtracking search very frequently finds a context that falsifies the conjecture. But it is of little help to the user to know only that some error exists somewhere, with no hint where it is. Thus, when a conjecture is invalid, it is critical to present the reason to the user in an intelligible way.

6.1. ERROR CONTEXT REPORTING.   Since *Sat* maintains a context representing a conjunction of literals that characterizes the current case, one simple way of

reporting failed proofs is to print that conjunction of literals to the user. We call this *error context reporting*. For example, the invalidity of the conjecture $x \geq 0 \Rightarrow x > 10$, would be reported with the *error context $x \geq 0 \wedge x \leq 10$*. We also call this error context a *counterexample* even though a specific value for x is not supplied. We don't include quantified formulas in the reported error context, but only basic literals (§5.3.1).

One difficulty with error context reporting is that of producing a readable textual description of the error context, the *context printing problem*. If the literals are simply printed one by one, the list may become voluminous and include many redundant literals. To avoid this, Greg Nelson suggested in his thesis [Nelson 1981] that a succinct error context somehow be computed directly from the E-graph and Simplex tableau. In the case of the E-graph, Nelson [1981, Sec. 11] gave an attractive algorithm for finding a minimum-sized set of equalities equivalent to a given E-graph, and this algorithm is in fact implemented in Simplify. Simplify also implements an algorithm for translating the Simplex tableau into a succinct conjunction of literals, but it is not as elegant as the algorithm for the E-graph.

The difficulties of the context printing problem are exacerbated by the background predicate described in Section 2: the previously mentioned succinct representations of the E-graph and the Simplex tableau may include many literals that are redundant in the presence of the background predicate. Therefore, by default, Simplify *prunes* the error context by testing each of its literals and deleting any that are easily shown to be redundant in the presence of the background predicate and the previously considered literals. This is expensive, and we often use the switch that disables pruning. Indeed, now that we have implemented error localization, described in the next section, we sometimes use the switch that disables error context reporting altogether.

6.2. ERROR LOCALIZATION. The idea of *error localization* is to attribute the invalidity of the conjecture to a specific portion of its text.

For a conjecture like $x \geq 0 \Rightarrow x > 10$, error context reporting works well. But for the problems that occur in program checking, error contexts are often large even after pruning. If a ten-page conjecture includes the unprovable conjunct $i \geq 0$, a ten-page error context that includes the formula $i < 0$ is not a very specific indication of the problem. We would rather have an output like "Can't prove the postcondition $i \geq 0$ on line 30 of page 10". ESC relies on Simplify's error localization to report the source location and type of the program error that led to the invalid verification condition.

At first, error localization may seem hopelessly underspecified: is the invalidity of an implication to be blamed on the weakness of its antecedent or the strength of its consequent? To make error localization work, we extended Simplify's input language with *labels*. A label is just a text string; if P is a first-order formula and L is a label, then we introduce the notation $L : P$ as a first-order formula whose semantics are identical to P but that has the extra operational aspect that if Simplify refutes a conjecture containing an occurrence of $L : P$, and P is true in the error context, and P's truth is "relevant" to the error context, then Simplify will include the label L in its error report.

We also write $L^* : P$ as short for $\neg (L : \neg P)$, which is semantically equivalent to P but causes L to be printed if P is false and its falsehood is relevant. (Simplify doesn't actually use this definition; internally, it treats L^* as a primitive, called a negative label. But in principle, one primitive suffices.)

For example, suppose that a procedure dereferences the pointer p at line 10 and accesses $a[i]$ on line 11. The obvious verification condition has the form

$$Precondition \Rightarrow p \neq \mathbf{null} \wedge i \geq 0 \wedge \cdots \quad .$$

Using labels (say, `|Null@10|` and `|IndexNegative@11|`), the verification condition instead has the form:

$Precondition \Rightarrow$
$\quad$ `|Null@10|`$^* : p \neq \mathbf{null} \wedge$ `|IndexNegative@11|`$^* : i \geq 0 \wedge \cdots \quad .$

Thus, if the proof fails, Simplify's output will include a label whose name encodes the source location and error type of a potential error in the source program. This is the basic method ESC uses for error reporting. Todd Millstein has extended the method by changing the ESC verification condition generator so that labels emitted from failed proofs determine not only the source location of the error, but also the dynamic path to it [Millstein 1999]. We have also found labels to be useful for debugging failed proofs when using Simplify for other purposes than ESC.

Before we implemented labels in Simplify, we achieved error localization in ESC by introducing extra propositional variables called *error variables*. Instead of the labels `|Null@10|` and `|IndexNegative@11|` in the example above, we would have introduced error variables, say `|EV.Null@10|` and `|EV.IndexNegative@11|` and generated the verification condition

$Precondition \Rightarrow$
$\quad$ (`|EV.Null@10|` $\vee\ p \neq \mathbf{null}$) $\wedge$ (`|EV.IndexNegative@11|` $\vee\ i \geq 0$) $\wedge \ldots \quad .$

If the proof failed, the error context would set at least one of the error variables to **false**, and the name of that variable would encode the information needed to localize the error. We found, however, that error variables interfered with the efficacy of subsumption (§3.2) by transforming atomic formulas (§2) into nonatomic formulas, and interfered with the efficacy of the status test by transforming distinct occurrences of identical formulas into distinct formulas.

We have said that a label will be printed only if the formula that it labels is "relevant". For example, suppose that the query is

$$(P \wedge (Q \vee L_1 : R)) \vee (U \wedge (V \vee L_2 : R))$$

and that it turns out to be satisfiable by taking U and R to be true and P to be false. We would like L_2 to be printed, but we consider that it would be misleading to print L_1.

Perhaps unfortunately, we do not have a precise definition of "relevant". Operationally, Simplify keeps labels attached to occurrences of literals (including propositional proxy literals (§3.3)) and quantifier proxies (§5.3.1)), and when an error context is discovered, the positive labels of the asserted literals and the negative labels of the denied literals are printed.

An ancillary benefit of the label mechanism that we have found useful is that Simplify can be directed to print a log of each label as the search encounters it. This produces a dynamic trace of which proof obligation Simplify is working on at any moment. For example, if a proof takes a pathologically long time, the log will reveal which proof obligation is causing the problem.

So much for the basic idea. We now mention some details.

First, labels interact with canonicalization (§3.3). If two formulas are the same except for labels, canonicalizing them to be the same will tend to improve efficiency, since it will improve the efficacy of the status test.

Simplify achieves this in the case that the only difference is in the outermost label. That is, $L : P$ and $M : P$ will be canonicalized identically even if L and M are different labels. However, Simplify does not canonicalize two formulas that are the same except for different labels that are nested within them. For example, $(L : P) \Rightarrow Q$ might be canonicalized differently than $(M : P) \Rightarrow Q$. We don't have any data to know whether this is hurting performance.

Second, labels must be considered when rewriting Boolean structure, for example, when creating matching rules from quantified formulas. For example, we use the rewriting rule

$$P \wedge L : \textbf{true} \longrightarrow L : P.$$

If a conjunct simplifies to **true**, the rewrite rules obviously delete the conjunct. If the conjunct is labeled, we claim that it is appropriate to move the label to the other conjunct. Lacking a precise definition of relevance, we can justify this claim only informally: If, after the rewriting, P turns out to be true and relevant, then before the rewriting, the occurrence of **true** was relevant; hence L should be printed. If, after the rewriting, P turns out to be false or irrelevant, then before the rewriting, the occurrence of **true** was irrelevant. Similarly, we use the rules

$$P \vee L : \textbf{true} \longrightarrow L : \textbf{true}$$
$$P \vee L^* : \textbf{false} \longrightarrow L^* : P$$
$$P \wedge L^* : \textbf{false} \longrightarrow L^* : \textbf{false}.$$

More difficult problems arise in connection with the treatment of labels when Simplify puts bodies of quantified formulas into CNF and when it instantiates matching rules. Consider what happens when Simplify asserts the formula

$$(\forall x : P(x) \vee L : ((Q_1(x) \wedge Q_2(x)) \vee (R_1(x) \wedge R_2(x)))), \tag{10}$$

If formula (10) did not include the label L, Simplify would would produce four matching rules, corresponding to the following four formulas:

$$\begin{aligned} &(\forall x : P(x) \vee Q_1(x) \vee R_1(x)) \\ &(\forall x : P(x) \vee Q_1(x) \vee R_2(x)) \\ &(\forall x : P(x) \vee Q_2(x) \vee R_1(x)) \\ &(\forall x : P(x) \vee Q_2(x) \vee R_2(x)). \end{aligned} \tag{11}$$

With the label L present, Simplify produces the same four matching rules, but with portions of their bodies labeled in a manner that allows Simplify to output the label L when appropriate. In particular, Simplify will output L when the (heuristically relevant) formulas asserted in an error context include both $Q_1(T)$ and $Q_2(T)$ or both $R_1(T)$ and $R_2(T)$ for some term T. On the other hand, the inclusion of, for example, $Q_1(T)$ and $R_2(T)$ in an error context will not be sufficient to cause Simplify to output the label L, nor will the inclusion of $Q_1(T)$ and $Q_2(U)$, where U is a term not equivalent to T.

Simplify achieves this effect by synthesizing *derived labels* in addition to the *original labels* present in the input conjecture. We will not describe derived labels in full detail. As an example that gives the flavor of the approach, we mention the

rule for distributing $\vee$ into a labeled conjunction:

$$P \vee L : (Q \wedge R) \rightarrow (P \vee L\alpha : Q) \wedge (P \vee L\beta : R)$$

The label L is *reportable* if both of the *conjunct derived labels $L\alpha$ and $L\beta$* are reportable. (This is the clause of the recursive definition of "reportable" applicable to conjunct derived labels.) The effect of the rewrite rule is thus that L is reported if $Q \wedge R$ is true and relevant, which is what is desired. Simplify also uses *disjunct derived labels* and *parametrized derived labels*.

6.3. MULTIPLE COUNTEREXAMPLES.   As remarked in Section 3.2, Simplify can find multiple counterexamples to a conjecture, not just one. In practice, even a modest-sized invalid conjecture may have a very large number of counterexamples, many of them differing only in ways that a user would consider uninteresting. Reporting them all would not only be annoying, but would hide any interesting variability amidst an overwhelming amount of noise. Therefore, Simplify defines a subset of the labels to be *major labels* and reports only counterexamples that differ in all their major labels.

In more detail, Simplify keeps track of the set of all major labels already reported with any counterexample for the current conjecture, and whenever a literal with a label in this set is asserted (more precisely, asserted if the label is positive or denied if the label is negative), Simplify backtracks, just as if the context had been found to be inconsistent. In addition, an environment variable specifies a limit on the number of error contexts the user is interested in, and Simplify halts its search for more error contexts whenever the limit has been reached. Simplify also halts after reporting any error context having no major labels, which has the effect of limiting the number of error contexts to one in the case of unlabeled input.

When Simplify is run from the shell, the limit on the number of reported counterexamples defaults to 1. ESC/Java changes the default to 10 and generates conjectures in such a way that each counterexample will have exactly one major label, namely the one used to encode the type and location of a potential program error, such as `|Null@10|` or `|IndexNegative@11|` from the example in Section 6.2. These labels all include the character "@", so, in fact, in general Simplify defines a major label to be one whose name includes an "@".

7. *The E-graph in Detail*

In Section 4.2, we introduced the E-graph module with which Simplify reasons about equality. In this section we describe the module in more detail. The key ideas of the congruence closure algorithm were introduced by Downey et al. [1980], but this section describes the module in more detail, including propagating equalities, handling distinctions (§2), and undoing merges. First, we introduce the features and invariants of the basic E-graph data structure, and then we give pseudocode for the main algorithms.

7.1. DATA STRUCTURES AND INVARIANTS.   The E-graph represents a set of terms and an equivalence relation on those terms. The terms in the E-graph include all those that occur in the conjecture, and the equivalence relation of the E-graph equates two nodes if the equality of the corresponding terms is a logical consequence of the current equality assertions.
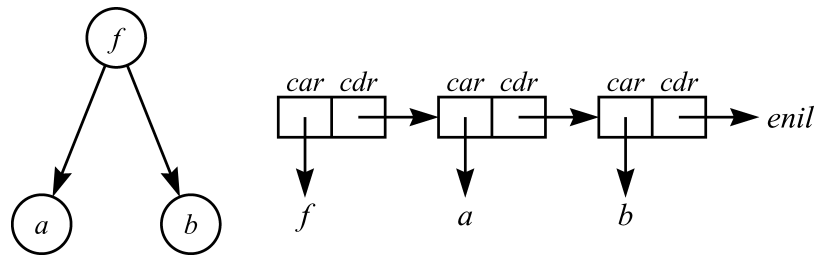
FIG. 3.   The term $f(a, b)$ as represented in an abstract term DAG on the left and in a concrete E-graph on the right.

We could use any of the standard methods for representing an equivalence relation, which are generally known as union-find algorithms. From these methods we use the so-called "quick find" approach. That is, each E-node p contains a field *p.root* that points directly at the canonical representative of p's equivalence class. Thus E-nodes p and q are equivalent exactly when *p.root* = *q.root*. In addition, each equivalence class is linked into a circular list by the *next* field. Thus, to merge the equivalence classes of p and q, we (1) reroot one of the classes (say, p's) by traversing the circular list p, *p.next*, *p.next.next*, ... updating all *root* fields to point to *q.root* and then (2) splice the two circular lists by exchanging *p.next* with *q.next*. For efficiency, we keep track of the size of each equivalences class (in the *size* field of the root) and reroot the smaller of the two equivalence classes. Although the union-find algorithm analyzed by Tarjan [1975] is asymptotically more efficient, the one we use is quite efficient both in theory and in practice [Knuth and Schönhage 1978], and merges are easy to undo, as we describe below.

Equality is not just any equivalence relation: it is a congruence. The algorithmic consequence is that we must represent a *congruence-closed* equivalence relation. In Section 4.2, we defined the congruence closure of a relation on the nodes of a vertex-labeled oriented directed graph with nodes of arbitrary out-degree. In the implementation, we use the standard reduction of a vertex-labeled oriented directed graph of general degree to an oriented unlabeled directed graph where every node has out-degree two or zero. That is, to represent the abstract term DAG (§4.2), in which nodes are labeled and can have any natural number of out-edges, we use a data structure that we call the *concrete E-graph*, in which nodes (called E-nodes) are unlabeled and either have out-degree two (*binary* E-nodes) or zero (*atomic* E-nodes). We learned this kind of representation from the programming language Lisp, so we will call the two edges from a binary E-node by their Lisp names, *car* and *cdr*. The basic idea for limiting the outdegree to two is to represent the sequence of children (§4.2) of a DAG node by a list linked by the *cdr* fields, as illustrated in Figure 3.

Here is a precise definition of the concrete E-graph representation:

Definition 1.   An E-node can *represent* either a function symbol, a term, or a list of terms. The rules are as follows: (1) Each function symbol f is represented by a distinct atomic E-node, $\lambda(f)$. (2) Each term $f(a_1, \ldots, a_n)$ is represented by a binary E-node e such that $e.car = \lambda(f)$ and $e.cdr$ represents the list $a_1, \ldots, a_n$. (3) A nonempty term list $(a_1, \ldots a_n)$ is represented by a binary E-node e such that $e.car$ is equivalent to some node that represents the term a_1 and $e.cdr$ represents the list $(a_2, \ldots, a_n)$. The empty term list is represented by a special atomic E-node *enil*.

E-nodes of types (1), (2), and (3) are called *symbol nodes*, *term nodes*, and *list nodes*, respectively. The words "is equivalent to" in (3) reflect our convention from Section 4.2 that "represents" means "represents up to congruence".

In Section 4.2, we defined the word "E-node" to refer to the nodes of an abstract term DAG, which correspond to just the term nodes of the concrete E-graph. In this section, we will use the word "E-node" to refer to all three types of concrete E-nodes.

In the concrete E-graph, we define a binary node p to be a *parent* of $p.car$ and of $p.cdr$. More precisely, it is a *car-parent* of the former and a *cdr-parent* of the latter. A node p is a *parent*, *car-parent*, or *cdr-parent* of an equivalence class if it is such a parent of some node in the class. Because every *cdr* field represents a list, and every *car* field represents either a term or a function symbol, we also have the following invariant

Invariant (CAR-CDR DICHOTOMY). A symbol node or term node may have *car*-parents, but not *cdr*-parents; a list node may have *cdr*-parents, but not *car*-parents.

In the concrete E-graph, we define two binary nodes to be congruent with respect to a relation R if their *car*'s and *cdr*'s are both related by R. We say that equivalence relation R is *congruence-closed* if any two nodes that are congruent under R are also equivalent under R. The *congruence closure* of a relation R on the nodes of the E-graph is the smallest congruence-closed equivalence relation that extends R. We leave it to the reader to verify that if R is a relation on term nodes and Q the congruence closure of R in the sense just defined, then Q restricted to term nodes is the congruence closure of R in the sense of Section 4.2.

The matching iterators and optimizations of Section 5.4 were presented there in terms of the abstract term DAG. Translating them to use the concrete E-graph is for the most part straightforward and will be left to the reader. One point that we do mention is that Simplify maintains mod-times (§5.4.2) for list nodes as well as for term nodes, and therefore *UpdateMT* iterates over concrete parents.

Because all asserted equalities are between term nodes, all congruences are either between two term nodes or between two list nodes, and therefore we have the following invariant:

Invariant (TERM-LIST-FSYM TRICHOTOMY). Every equivalence class in the E-graph either (1) consists entirely of term nodes, (2) consists entirely of list nodes, or (3) is a singleton of the form $\{\lambda(f)\}$ for some function symbol f.

The singleton class $\{enil\}$ is of the form (2).

Returning to the implementation of congruence closure, the reduction to the concrete E-graph means that we need to worry only about congruences between nodes of degree 2. We define the *signature* of a binary node p to be the pair $(p.car.root.id, p.cdr.root.id)$, where the *id* field is a numerical field such that distinct nodes have distinct *id*'s. The point of this definition is that two binary nodes are congruent if and only if they have the same signature.

Merging two equivalence classes P and Q may create congruences between parents of nodes in P and parents of nodes in Q. To detect such new congruences, it suffices to examine all parents of nodes in one of the classes and test whether they participate in any new congruences. We now discuss these two issues, the test and the enumeration.

To support the test for new congruences, we maintain the *signature table $sigTab$*, a hash table with the property that, for any signature (i, j), $sigTab(i, j)$ is an E-node

with signature (i, j), or is **nil** if no such E-node exists. When two equivalence classes merged, the only nodes whose signatures change are the parents of whichever class does not give its *id* to the merged class. The key idea in the Downey–Sethi–Tarjan algorithm is to use this fact to examine at most half of the parents of the merged equivalence class. This can be done by keeping a list of parents of each equivalence class along with the size of the list, and by ensuring that the *id* of the merged class is the *id* of whichever class had more parents.

We represent the parent lists "endogenously", that is, the links are threaded within the nodes themselves, as follows: Each E-node e contains a *parent* field. For each root node r, *r.parent* is some parent of r's equivalence class, or is **nil** if the class has no parent. Each binary E-node p contains two fields *samecar* and *samecdr*, such that for each equivalence class Q, all the *car*-parents of Q are linked into a circular list by the *samecar* field, and all the *cdr*-parents of Q are linked into a circular list by the *samecdr* field.

With these data structures, we can implement the iteration

> **for each** parent p of the equivalence class of the root r **do**
>   Visit p
> **end**

as follows:

> **if** *r.parent* $\neq$ **nil then**
>   **if** r is a symbol node or term node then
>   **for each** node p in the circular list
>   *r.parent, r.parent.samecar, r.parent.samecar.samecar, . . .* **do**
>   Visit p
>   **end**
>   **else**
>   **for each** node p in the circular list
>   *r.parent, r.parent.samecdr, r.parent.samecdr.samecdr, . . .* **do**
>   Visit p
>   **end**
>   **end**

To support the key idea in the Downey–Sethi–Tarjan algorithm, we introduce the *parentSize* field. The *parentSize* field of the root of an equivalence class contains the number of parents of the class.

The facts represented in the E-graph include not only equalities but also distinctions (both binary and general (§2)). To represent distinctions in the E-graph, we therefore add structure to the E-graph to represent that certain pairs of nodes are *unmergeable* in the sense that if they are ever combined into the same equivalence class, the context is unsatisfiable.

Simplify's E-graph uses two techniques to represent unmergeability, one suited to binary distinctions and the other to general distinctions.

To represent binary distinctions, we supply each E-node e with a list of E-nodes, *e.forbid*, called a *forbid list*, and maintain the following invariant:

Invariant (FORBID LIST VALIDITY). If a binary distinction between two E-nodes x and y has been asserted on the current path, then *x.root.forbid* contains a node equivalent to y and *y.root.forbid* contains a node equivalent to x.

To assert a binary distinction $X \neq Y$, we find the E-nodes x and y that represent X and Y. If they have the same root, then the assertion produces an immediate contradiction. Otherwise, we add x to *y.root.forbid* and also add y to *x.root.forbid*.

When asserting an equality $X = Y$, we find the E-nodes x and y that represent X and Y and traverse the forbid list of either $x.root$ or $y.root$ (whichever list is shorter), looking for nodes equivalent to the other. If one is found, then the assertion produces an immediate contradiction. If no contradiction is found, we set the forbid list of the root of the combined equivalence class to be the concatenation of the two lists.

Because an E-node may be on more than one forbid list, we represent forbid lists "exogenously": that is, the nodes of the list are distinct from the E-nodes themselves. Specifically, forbid lists are represented as circular lists linked by a *link* field and containing E-nodes referenced by an *e* field.

Since an n-way general distinction is equivalent to the conjunction of $O(n^2)$ binary distinction, it seems unattractive to use forbid lists for representing general distinctions, so for these we introduce a technique called *distinction classes*. A distinction class is a set of E-nodes any two of which are unmergeable. A new distinction class is allocated for each asserted general distinction. To represent membership in distinction classes, we supply each E-node e with a bit vector $e.distClasses$ and maintain the invariant:

Invariant (DISTINCTION CLASS VALIDITY). For any root E-node r and integer i, $r.distClasses[i]$ is **true** if some E-node in r's equivalence class is a member of the ith distinction class created on the current path.

To assert a general distinction $\text{DISTINCT}(T_1, \ldots, T_n)$, we find an E-node t_i representing T_i. If any two of these E-nodes have the same root, then the assertion produces an immediate contradiction. Otherwise, we allocate a new distinction class number d and set bit d of each $t_i.root.distClasses$ to **true**.

When asserting an equality $X = Y$, we find the E-nodes x and y that represent X and Y. If $x.root.distClasses$ and $y.root.distClasses$ contain any **true** bit in common, then the assertion produces an immediate contradiction. If no contradiction is found, we set the *.distClasses* field of the root of the combined equivalence class to be the union (bitwise OR) of $x.root.distClasses$ and $y.root.distClasses$

A fine point: Simplify allocates at most k distinction classes on any path in its backtracking search, where k is the number of bits per word in the host computer. For any additional occurrences of general distinctions, it retreats to the expansion into $\binom{n}{2}$ binary distinctions. If this retreat had become a problem, we were prepared to use multiword bit vectors for *distClasses*, but in our applications, running on our 64-bit Alpha processors, the retreat never became a problem.

In summary, here is the implementation of the test for unmergeability:

```
proc Unmergeable(x, y : E-node) : boolean ≡
  var p, q, pstart, qstart, pptr, qptr in
    p, q := x.root, y.root;
    if p = q then return false end;
    if p.distClasses ∩ q.distClasses ≠ { } then
      return true
    end;
    pstart, qstart := p.forbid, q.forbid;
    if pstart ≠ nil and qstart ≠ nil then
      pptr, qptr := pstart, qstart;
    loop
      if pptr.e.root = q or qptr.e.root = p then
        return true
      else
        pptr, qptr := pptr.link, qptr.link
      end
```

```
        if pptr = pstart or qptr = qstart then exit loop end
      end;
      return false
    end
  end
```

For a variety of reasons, we maintain in the E-graph an explicit representation of the relation "is congruent to." This is also an equivalence relation and again we could use any standard representation for equivalence relations, but while the main equivalence relation of the E-graph is represented in the *root*, *next*, and *size* fields using the quick-find technique, the congruence relation is represented with the single field *cgPtr* using simple, unweighted Fischer–Galler trees [Galler and Fischer 1964; Knuth 1968, Sec. 2.3.3]. That is, the nodes are linked into a forest by the *cgPtr* field. The root of each tree of the forest is the canonical representative of the nodes in that tree, which form a congruence class. For such a *congruence root* r, $r.cgPtr = r$. For a nonroot x, $x.cgPtr$ is the tree parent of x.

In Section 5.4.1, which defined the iterators for matching triggers in the E-graph, we mentioned that for efficiency it was essential to ignore nodes that are not canonical representatives of their congruence classes, this is easily done by testing whether $x.cgPtr = x$. We will also maintain the invariant that the congruence root of a congruence class is the node that represents the class in the signature table. That is,

Invariant (SIGNATURE TABLE CORRECTNESS). The elements of *sigTab* are precisely the pairs $((x.car.root.id, x.cdr.root.id), x)$ such that x is a binary E-node that is a congruence root.

The last subject we will touch on before presenting the implementations of the main routines of the E-graph module is to recall from Section 4.4 the strategy for propagating equalities from the E-graph to the ordinary theories. For each ordinary theory T, we maintain the invariant:

Invariant (PROPAGATION TO T). Two equivalent E-nodes have non-nil *T_unknown* fields if and only if these two *T.Unknown*'s are equated by a chain of currently propagated equalities from the E-graph to the T module.

When two equivalence classes are merged in the E-graph, maintaining this invariant may require propagating an equality to T. To avoid scanning each equivalence class for non-nil *T_unknown* fields, the E-graph module also maintains the invariant:

Invariant (ROOT *T_unknown*). If any E-node in an equivalence class has a non-nil *T_unknown* field, then the root of the equivalence class has a non-nil *T_unknown* field.

We now describe the implementations of some key algorithms for maintaining the E-graph, namely those for asserting equalities, undoing assertions of equalities, and creating E-nodes to represent terms.

7.2. ASSERTEQ AND MERGE. To assert the equality $X = Y$, Simplify calls *AssertEQ(x, y)* where x and y are E-nodes representing the terms X and Y. *AssertEQ* maintains a work list in the global variable *pending*. This is a list of pairs of E-nodes whose equality is implicit in the input but not yet represented in the E-graph. *AssertEQ* repeatedly merges pairs from the pending list, checking for unmergeability before calling the procedure *Merge*.

```
proc AssertEQ(x, y : E-node) ≡
  pending := {(x, y)};
  while pending ≠ {} do
    remove a pair (p, q) from pending;
    p, q := p.root, q.root;
    if p ≠ q then
      // p and q are not equivalent.
      if Unmergeable(p, q) then
        refuted := true;
        return
      else
        Merge(p, q)
      end
    end
  end
  // The E-graph is congruence-closed.
end
```

The call $Merge(x, y)$ combines the equivalence classes of x and y, adding pairs of nodes to *pending* as new congruences are detected.

$Merge(x, y)$ requires that x and y are roots, that they are neither equivalent nor unmergeable, and that either (1) they are both term nodes representing terms whose equality is implicit in the current context, or (2) they are list nodes representing lists of terms $(X_1, \ldots, X_n)$ and $(Y_1, \ldots, Y_n)$ such that each equality $X_i = Y_i$ is implicit in the current context.

```
     proc Merge(x, y : E-node) ≡

         // Step 1: Make x be the root of the larger equivalence class.
M1       if x.size < y.size then
M2         x, y := y, x
M3       end;
         // x will become the root of the merged class.

         // Step 2: Maintain invariants relating E-graph to ordinary theories.
M4       for each ordinary theory T do
           // maintain PROPAGATION TO T
M5         if x.T_unknown ≠ nil and y.T_unknown ≠ nil then
M6           Propagate the equality of x.T_unknown with y.T_unknown to T
M7         end
           // maintain ROOT T_unknown
M8         if x.T_unknown = nil then x.T_unknown := y.T_unknown end;
M9       end;

         // Step 3: Maintain sets for pattern-element optimizations (§5.4.3).
M10      UpdatePatElems(x, y);

         // Step 4: Make x unmergable with nodes now unmergable with y.
         // Merge forbid lists.
M11      if y.forbid ≠ nil then
M12        if x.forbid = nil then
M13          x.forbid := y.forbid
M14        else
M15          x.forbid.link, y.forbid.link := y.forbid.link, x.forbid.link
M16        end
M17      end;
         // Merge distinction classes.
```

```
M18      x.distClasses := x.distClasses ∪ y.distClasses;

         // Step 5: Update sigTab, adding pairs of newly congruent nodes to
         // pending.
M19      var w in
             // 5.1: Let w be the root of the class with fewer parents.
M20          if x.parentSize < y.parentSize then
M21              w := x
M22          else
M23              w := y
M24          end;
             // (Only parents of w's class will get new signatures.)
             // 5.2: Remove old signatures of w's class's parents.
M25          for each parent p of w's equivalence class do
M26              if p = p.cgPtr then // p is originally a congruence root.
M27                  Remove ((p.car.root.id, p.cdr.root.id), p) from the signature table
M28              end;
             // 5.3: Union the equivalence classes.
             // 5.3.1: Make members of y's class point to new root x.
M29          for each v in the circular list y, y.next, y.next.next, … do
M30              v.root := x
M31          end;
             // 5.3.2: Splice the circular lists of equivalent nodes
M32          x.next, y.next := y.next, x.next;
M33          x.size := x.size + y.size;
             // 5.4: Preserve signatures of the larger parent set by swapping
             // id's if necessary.
M34          if x.parentSize < y.parentSize then
M35              x.id, y.id := y.id, x.id
M36          end;
             // 5.5: Put parents of w into sigTab with new signatures,
             // and add pairs of newly congruent pairs to pending.
M37          for each parent p of w's equivalence class do
M38              if p = p.cgPtr then // p is originally a congruence root.
M39                  if the signature table contains an entry q under the key
M40                      (p.car.root.id, p.cdr.root.id) then
                         // Case 1: p joins q's congruence class.
M41                      p.cgPtr := q;
M42                      pending := pending ∪ {(p, q)}
M43                  else
                         // Case 2: p remains a congruence root.
M44                      Insert ((p.car.root.id, p.cdr.root.id), p) into the signature table
M45                  end
M46              end
M47          end
M48      end;

         // Step 6: Merge parent lists.
M49      if y.parent ≠ nil then
M50          if x.parent = nil then
M51              x.parent := y.parent
M52          else
                 // Splice the parent lists of x and y.
M53              if x and y are term nodes then
M54                  x.parent.samecar, y.parent.samecar :=
M55                      y.parent.samecar, x.parent.samecar
M56              else
M57                  x.parent.samecdr, y.parent.samecdr :=
M58                      y.parent.samecdr, x.parent.samecdr
```

```
M59          end
M60        end
M61    end;
M62    x.parentSize := x.parentSize + y.parentSize;

       // Step 7: Update mod-times for mod-time matching optimization (§5.4.2).
M63    UpdateMT(x)

       // Step 8: Push undo record.
M64    Push onto the undo stack (§3.1) the pair ("UndoMerge", y)
M65 end
```

The union of *distClasses* fields on line M18 is implemented as a bitwise OR operation on machine words. The **for each** loops starting on lines M25 and M37 are implemented as described in Section 7.1.

7.3. UNDOMERGE.   During backtracking, when we pop an entry of the form ("UndoMerge", y) from the undo stack, we must undo the effect of *Merge* by calling *UndoMerge*(y).

Here is the pseudocode for *UndoMerge*, followed by some explanatory remarks. In what follows, when we refer to "the merge" or to a step in the execution of *Merge*, we mean the particular execution of *Merge* whose effect is being undone.

```
       proc UndoMerge(y : E-node) ≡

             // Compute the other argument of Merge.
U1     var x := y.root in

             // Undo Step 6 of Merge.
U2        x.parentSize := x.parentSize − y.parentSize;
U3        if y.parent ≠ nil then
U4          if x.parent = y.parent then
U5            x.parent := nil
U6          else
                // Unsplice the parent lists of x and y.
U7            if x and y are term nodes then
U8              x.parent.samecar, y.parent.samecar :=
U9                  y.parent.samecar, x.parent.samecar
U10           else
U11             x.parent.samecdr, y.parent.samecdr :=
U12                 y.parent.samecdr, x.parent.samecdr
U13           end
U14         end
U15       end;

             // Undo Step 5 of Merge.
U16       var w in
U17         if x.parentSize < y.parentSize then
U18           w := x
U19         else
U20           w := y
U21         end;
             // w now has the value computed in lines M20–M24.
             // Undo Case 2 iterations in Step 5.5 of Merge.
U22         for each parent p of w's equivalence class do
U23           if p = p.cgPtr then
U24             Remove ((p.car.root.id, p.cdr.root.id), p) from the signature table
```

```
U25                 end;
U26             end;
                // Undo Step 5.4 of Merge
U27             if x.parentSize < y.parentSize then
U28                 x.id, y.id := y.id, x.id
U29             end;
                // Undo Step 5.3 of Merge
U30             x.size := x.size − y.size;
U31             x.next, y.next := y.next, x.next;
U32             for each v in the circular list y, y.next, y.next.next, … do
U33                 v.root := y
U34             end;
                // Undo Step 5.2 and Case 1 iterations of Step 5.5 of Merge.
U35             for each parent p of w's equivalence class do
U36                 var cg := p.cgPtr in
U37                     if (p.car.root ≠ cg.car.root or p.cdr.root ≠ cg.cdr.root) then
                            // Undo a Case 1 iteration in Step 5.5 of Merge.
U38                         p.cgPtr := p
U39                     end;
U40                     if p.cgPtr = p then
                            // Undo an iteration of Step 5.2 of Merge.
U41                         Insert ((p.car.root.id, p.cdr.root.id), p) into the signature table
U42                     end
U43                 end
U44             end
U45         end;

                // Undo Step 4 of Merge.
U46         x.distClasses := x.distClasses − y.distClasses;
U47         if x.forbid = y.forbid then
U48             x.forbid := nil
U49         else if y.forbid ≠ nil then
U50             x.forbid.link, y.forbid.link := y.forbid.link, x.forbid.link
U51         end;

                // Undo Step 2 of Merge.
U52         for each ordinary theory T do
U53             if x.T_unknown = y.T_unknown then x.T_unknown := nil end
U54         end

U55     end
U56 end
```

Recall that the overall form of *Merge* is:

> **Step 1:** Make *x* be the root of the larger equivalence class.
> **Step 2:** Maintain invariants relating E-graph to ordinary theories.
> **Step 3:** Maintain sets for pattern-element optimizations.
> **Step 4:** Make *x* unmergable with nodes now unmergable with *y*.
> **Step 5:** Update *sigTab*, adding pairs of newly congruent nodes to *pending*.
> **Step 6:** Merge parent lists.
> **Step 7:** Update mod-times for mod-time matching optimization
> **Step 8:** Push undo record.

Step 1 has no side effects; it merely sets *x* to the E-node that will be the root of the merged equivalence class, and *y* to be the root of the other pre-merge equivalence

class. *UndoMerge* thus has no actual undoing code for Step 1. However, line U1 ensures that x and y have the values that they had after line M3.

UndoMerge contains no code to undo the calls to *UpdatePatElems* and *UpdateMT* (Steps 3 and 7 of *Merge*). These routines create their own undo records, as described in Sections 5.4.3 and 5.4.2. Since the undoing code for these records is independent of the rest of *UndoMerge* (and vice-versa), it doesn't matter that these undo records are processed out of order with the rest of *UndoMerge*.

Finally, *UndoMerge* has no code for undoing Step 8, since Pop will have already popped the undo record from the undo stack before calling *UndoMerge*.

The remaining side effects of *Merge* are undone mostly in the reverse order that they were done, except that the substeps of Step 5 are undone slightly out of order, as discussed below. The process is mostly straightforward, but we call attention to a few points that might not be immediately obvious.

In the undoing code for Step 6, we rely on the fact that $x.parent$ and $y.parent$ could not have been equal before the merge. The precondition for $Merge(x, y)$ requires that x and y must either both be term nodes or both be list nodes. If they are term nodes, then $x.parent.car$ is equivalent to x and $y.parent.car$ is equivalent to y. Since the precondition for $Merge(x, y)$ requires that x and y not be equivalent, it follows that $x.parent \neq y.parent$. The case where x and y are list nodes is similar.

In the undoing code for Step 4, we rely on the fact that $x.distClasses$ and $y.distClasses$ must have been disjoint before the merge, which follows from the specification that the arguments of *Merge* not be unmergeable. Thus, $x.distClasses$ can be restored (on line U46) by set subtraction implemented with bit vectors.

Similarly, in the undoing code for Step 2, we rely on the fact that $x.T\_unknown$ and $y.T\_unknown$ could not have been equal before the merge.

The most subtle part of *UndoMerge* is the undoing of Step 5 of *Merge*. Recall that the structure of Step 5 is:

5.1: Let w be the root of the class with fewer parents.
5.2: Remove old signatures of w's class's parents.
5.3: Union the equivalence classes.
5.4: Preserve signatures of the larger parent set by swapping
  id's if necessary.
5.5: Put parents of w into *sigTab* with new signatures,
  and add pairs of newly congruent pairs to *pending*.

Step 5.1 has no nonlocal side effects, so needs no undoing code. However, lines U17–U21 of *UndoMerge* ensure that w has the value that is had at the end of Step 5.1 of *Merge*, since this is needed for the undoing of the rest of Step 5. The remaining substeps of Step 5 are undone mostly in the reverse order that they were done, the exception being that a part of Step 5.5—namely, the linking together of congruence classes (line M41)—is undone somewhat later, for reasons that we now explain.

Properly undoing the effects of *Merge* upon congruence pointers depends on two observations. The only place that *Merge* updates a $cgPtr$ field is at line M41, $p.cgPtr := q$. When this is executed, (1) p is a congruence root before the merge, and (2) p and q become congruent as a result of the merge. Because of (1), the value that $p.cgPtr$ should be restored to is simply p. Because of (2), *UndoMerge* can locate those p's whose $cgPtr$ fields need to be restored by locating all parents p of w's equivalence class that are not congruent to $p.cgPtr$ in the prestate of the

merge. (*UndoMerge* enumerates these p's after restoring the equivalence relation to its pre-merge state.)

We conclude our discussion of *UndoMerge* with an observation about the importance of the Fischer–Galler trees in the representation of congruences classes. In our original implementation of Simplify's E-graph module, we did not use Fischer–Galler trees. In place of the SIGNATURE TABLE CORRECTNESS invariant given above, we maintained—or so we imagined—a weaker invariant requiring only that the signature table contain one member of each congruence class. The result was a subtle bug: After a *Merge-UndoMerge* sequence, all congruence classes would be still be represented in the signature table, but not necessarily with the same representatives they had before the *Merge*; a subsequent *UndoMerge* could then leave a congruence class unrepresented.

Here is an example illustrating the bug. Swapping of *id* fields plays no role in the example, so we will assume throughout that larger equivalence classes happen to have larger parent sets, so that no swapping of *id* fields occurs.

(1) We start in a state where nodes w, x, y, and z are the respective roots of different equivalence classes W, X, Y, and Z, and the binary nodes $(p, w), (p, x), (p, y)$, and (p, z) are present in the E-graph.

(2) W and X get merged. Node x becomes the root of the combined equivalence class WX, and node (p, w) is rehashed and found to be congruent to (p, x), so they are also merged.

(3) WX and Y get merged. Node y becomes the root of the combined equivalence class WXY, and node (p, x) is rehashed and found to be congruent to (p, y), so they are also merged.

(4) The merges of (p, x) with (p, y) and of WX with Y are undone. In the process of undoing the merge of WX with Y, the parents of WX are rehashed. Nodes (p, w) and (p, x) both have signature $(p.id, x.id)$. In the absence of the Fischer–Galler tree, whichever one happens to be rehashed first gets entered in the signature table. It happens to be (p, w).

(5) The merges of (p, w) with (p, x) and of W with X are undone. In the process of undoing the merge of W with X, the parents of W are rehashed. Node (p, w) is removed from the signature table and re-entered under signature $(p.id, w.id)$. At this point (p, x) has signature $(p.id, x.id)$, but the signature table contains no entry for that signature.

(6) X and Z get merged, x becomes the root of the combined equivalence class, and the parents of Z are rehashed. The new signature of (p, z) is $(p.id, x.id)$. Since the signature table lacks an entry for this signature, the congruence of (p, z) and (p, x) goes undetected.

7.4. CONS. Finally, we present pseudo-code for $Cons(x, y)$, which finds or constructs a binary E-node whose *car* and *cdr* are equivalent to x and y, respectively. We leave the other operations to the reader.

```
proc Cons(x, y : E-node) : E-node ≡
  var res in
    x := x.root;
    y := y.root;
    if sigTab(x.id, y.id) ≠ nil then
      res := sigTab(x.id, y.id)
```

```
else
   res := new(E-node);
   res.car := x;
   res.cdr := y;
   // res is in a singleton equivalence class.
   res.root := res;
   res.next := res;
   res.size := 1;
   // res has no parents.
   res.parent := nil;
   res.parentSize := 0;
   res.plbls := { };
   // Set res.id to the next available Id.
   res.id := idCntr;
   idCntr := idCntr + 1;
   // res is associated with no unknown.
   for each ordinary theory T do
      res.T_unknown := nil
   end;

   // res is a car-parent of x.
   if x.parent = nil then
      x.parent := res;
      res.samecar := res
   else
      // Link res into the parent list of x.
      res.samecar := x.parent.samecar;
      x.parent.samecar := res
   end;
   x.parentSize := x.parentSize + 1;

   // res is a cdr-parent of y.
   if y.parent = nil then
      y.parent := res;
      res.samecdr := res
   else
      // Link res into the parent list of y.
      res.samecdr := y.parent.samecdr;
      y.parent.samecdr := res
   end;
   y.parentSize := y.parentSize + 1;

   // res is the root of a singleton congruence class.
   res.cgPtr := res;
   Insert ((x.id, y.id), res) into sigTab;

   if x = λ(f) for some function symbol f then
      // res is an application of f.
      res.lbls := {f}
   else
      res.lbls := { }
   end;

   // res is inactive.
   res.active := false;

   Push onto the undo stack (§3.1) the pair ("UndoCons", res)
end;
```

```
            return res;
        end
    end
```

In the case where $x = \lambda(f)$, the reader may expect f to be added to *plbls* of each term E-node on the list y. In fact, Simplify performs this addition only when the new E-node is activated (§5.2), not when it is created.

7.5. THREE FINAL FINE POINTS.   We have three fine points to record about the E-graph module that are not reflected in the pseudo-code above.

First, the description above implies that any free variable in the conjecture will become a constant in the query (§2), which is represented in the E-graph by the *Cons* of a symbol node (§7.1) with *enil*. In fact, as an optimization to reduce the number of E-nodes, Simplify represents such a free variable by a special kind of term node whose representation is like that of a symbol node. Since the code above never takes the *car* or *cdr* of any term nodes except those known to be parents, the optimization has few effects on the code in the module, although it does affect the interning (§4.5) code and the code that constructs matching triggers (§5.1).

Second, because Simplify never performs matching during plunging (§4.6), there is no need to update mod-times (§5.4.2) and pattern element (§5.4.3) sets for merges that occur during plunging, and Simplify does not do so.

Third, recall from Section 2 that Simplify allows users to define quasi-relations with its DEFPRED facility. When an equivalence class containing **@true** is merged with one containing an application of such a quasi-relation, Simplify instantiates and asserts the body of the DEFPRED command. Simplify also notices most (but not all) cases in which an application of such a quasi-relation becomes unmergeable (§7.1) with **@true**, and in these cases it instantiates and asserts the negation of the body. Simplify avoids asserting multiple redundant instances of quasi-relation bodies by using a table of fingerprints (§5.2) (in essentially the same way that it avoids asserting redundant instances of matching rules).

8. *The Simplex Module in Detail*

In this section, we describe an incremental, resettable procedure for determining the satisfiability over the rationals of a conjunction of linear equalities and inequalities (§2). The procedure also determines the equalities between unknowns implied by the conjunction. This procedure is based on the Simplex algorithm. We also describe a few heuristics for the case of integer unknowns. The procedure and heuristics described in this section are the semantics of arithmetic built into Simplify.

The Simplex algorithm was invented by George Dantzig [Dantzig 1963] and is widely used in Operations Research [Chvatal 1983]. Our procedure shares with the Simplex algorithm a worst case behavior that would be unacceptable, but that does not seem to arise in practice.

We will describe the Simplex algorithm from scratch, as it is used in Simplify, rather than merely refer the reader to a standard operations research text. This lengthens our paper, but the investment of words seems justified, since Simplify requires an incremental, resettable, equality-propagating version of the algorithm and also because Simplify unknowns are not restricted to be non-negative, as they seem to be in most of the operations research literature on the algorithm. This section

is closely based on Section 12 of Nelson's revised Ph.D. thesis [Nelson 1981], but our presentation is smoother and corrects an important bug in Nelson's account. Our description is also somewhat discrepant from the actual code in Simplify, since we think it more useful to the reader to present the code as it ought to be instead of as it is.

Subsections 8.1, 8.2, 8.3, 8.4, 8.5, and 8.6 describe in order: the interface to the Simplex module, the tableau data structure used in the implementation, the algorithm for determining satisfiability, the modifications to it for propagating equalities, undoing modifications to the tableau, and some heuristics for integer arithmetic.

Before we dive into all these details, we mention an awkwardness caused by the limitation of the Simplex algorithm to linear equalities and inequalities. Because of this limitation, Simplify treats some but not all occurrences of the multiplication sign as belonging to the Simplex theory. For example, to construct the E-node for $2 \times x$ we ensure that this E-node and x's both have unknowns, and assert that the first unknown is double the second, just as described in Section 4.5. The same thing happens for $c \times x$ in a context where c is constrained to equal 2. However, to construct the E-node for $x \times y$ where neither x nor y is already constrained to equal a numerical constant, we give up, and treat $\times$ as an uninterpreted function symbol. Even if x is later equated with a numerical constant, Simplify, as currently programmed, continues to treat the occurrence of $\times$ as an uninterpreted function symbol.

In constructing AF's (§4.5), the Simplex module performs the canonicalizations hinted at in Section 4.5. For example, if the conjecture includes the subformulas $T_1 < T_2$, $T_2 > T_1$, $T_1 \geq T_2$, the literals produced from them will share the same AF, with the literal for the last having opposite sense from the others. However, $2 \times T_1 < 2 \times T_2$ will be canonicalized differently.

8.1. THE INTERFACE. Since *Simplex* is an ordinary theory, the interface it implements is the one described in Section 4.4. In particular, it implements the type *Simplex.Unknown*, representing an unknown rational number, and implements the abstract variables *Simplex.Asserted* and *Simplex.Propagated*, which represent respectively the conjunction of Simplex literals that have been asserted and the conjunction of equalities that have been propagated from the Simplex module. (In the remainder of Section 8, we will often omit the leading "*Simplex.*" when it is clear from context.)

As mentioned in Section 4.4, the fundamental routine in the interface is *AssertLit*, which tests whether a literal is consistent with *Asserted* and, if it is, conjoins the literal to *Asserted* and propagates equalities as necessary to maintain

Invariant (PROPAGATION FROM *Simplex*). For any two connected (§4.4) unknowns u and v, the equality $u = v$ is implied by *Propagated* iff it is implied by *Asserted*.

Below, we describe two routines in terms of which *AssertLit* is easily implemented. These two routines operate on *formal affine sums* of unknowns, that is formal sums of terms, where each term is either a rational constant or the product of a rational constant and an *Unknown*:

> **proc** *AssertGE*(*fas* : formal affine sum of connected unknowns);
> /∗ If *fas* ≥ 0 is consistent with *Asserted*, set *Asserted* to *fas* ≥ 0 ∧ *Asserted* and propagate
> equalities as required to maintain PROPAGATION FROM *Simplex*. Otherwise, set *refuted*
> to **true**. ∗/

> **proc** *AssertZ*(*fas* : formal affine sum of connected unknowns);
>
> /* If *fas* = 0 is consistent with *Asserted*, set *Asserted* to *fas* = 0 ∧ *Asserted* and propagate equalities as required to maintain PROPAGATION FROM *Simplex*. Otherwise, set *refuted* to **true**. */

The Simplex module exports the following procedures for associating Simplex unknowns (for the remainder of Section 8, we will simply say "unknowns") with E-nodes and for maintaining the abstract predicate *Asserted* over the unknowns:

> **proc** *UnknownForEnode*(*e* : *E-node*) : *Unknown*;
>
> /* Requires that *e.root* = *e*. Returns *e.Simplex_Unknown* if it is not **nil**. Otherwise sets *e.Simplex_Unknown* to a newly-allocated unknown and returns it. (The requirement that *e* be a root causes the invariants PROPAGATION TO *Simplex* and ROOT SIMPLEX UNKNOWN to be maintained automatically.) */

Of course, these routines must push undo records, and the module must provide the undoing code to be called by *Pop*.

8.2. THE TABLEAU. The *Simplex tableau* consists of

—three natural numbers m, n, and $dcol$, where $0 \le dcol \le m$,

—two one-dimensional arrays of Simplex unknowns, $x[1], \ldots, x[m]$ and $y[0], \ldots, y[n]$, and

—a two-dimensional array of rational numbers $a[0, 0], \ldots, a[n, m]$.

Each Simplex unknown u has a boolean field $u.restricted$. If $u.restricted$ is **true**, the unknown u is said to be *restricted*, which means that its value must be nonnegative.

We will present the tableau in the following form:

$$
\begin{array}{c|cccc}
 & 1 & x[1] & \cdots & x[m] \\
\hline
y[0] & a[0, 0] & a[0, 1] & \cdots & a[0, m] \\
\vdots & \vdots & \vdots & & \vdots \\
y[n] & a[n, 0] & a[n, 1] & \cdots & a[n, m]
\end{array}
$$

In the displayed form of the tableau restricted unknowns will be superscripted with the sign $\ge$, and a small ∇ will be placed to the right of column *dcol*.

The constraints represented by the tableau are the row constraints, dead column constraints, and sign constraints. Each row i of the tableau represents the *row constraint*

$$
y[i] = a[i, 0] + \sum_{j : 1 \le j \le m} a[i, j] \times x[j].
$$

For each j in the range $1, \ldots, dcol$, there is a *dead column constraint*, $x[j] = 0$. Columns with index $1, \ldots, dcol$ are the *dead* columns; columns $dcol + 1, \ldots, m$ are the *live* columns; and column 0 is the *constant* column. Finally, there is a *sign constraint* $u \ge 0$ for each restricted unknown. Throughout the remainder of Section 8, *RowSoln* denotes the solution set of the row constraints, *DColSoln* denotes the solution set of the dead column constraints, and *SignSoln* denotes the solution set of the sign constraints. We refer to the row constraints and the dead column constraints collectively as *hyperplane constraints* and denote their solution set as *HPlaneSoln*. We refer to the hyperplane constraints and the sign constraints

collectively as *tableau constraints* and denote their solution set as *TableauSoln*. We will sometimes identify these solution sets with their characteristic predicates.

It will be convenient to have a special unknown constrained to equal 0. Therefore, the first row of the tableau is owned by a special unknown that we call *Zero*, whose row is identically 0.

$$y[0] = Zero$$
$$a[0, j] = 0, \text{ for } j : 0 \leq j \leq m$$

The unknown *Zero* is allocated at initialization time and connected to the E-node for 0.

Here is a typical tableau in which *dcol* is 0.

	1	$^\nabla m^\geq$	$n^\geq$	p
Zero	0	0	0	0
$w^\geq$	0	1	-1	2
z	-1	-1	-3	0

$$(12)$$

The tableau (12) represents the tableau constraints:

$$m \geq 0$$
$$n \geq 0$$
$$w \geq 0$$
$$Zero = 0$$
$$w = m - n + 2p$$
$$z = -m - 3n - 1.$$

The basic idea of the implementation is that the tableau constraints are the concrete representation of the abstract variable *Asserted*, which is the conjunction of Simplex literals that have been asserted. There is one fine point: All unknowns that occur in arguments to *AssertLit* are connected to E-nodes, but the implementation allocates some unknowns that are not connected to E-nodes (these are the classical "slack variables" of linear programming). These must be regarded as existentially quantified in the representation:

Asserted = (∃*unc* : *TableauSoln*)
        where *unc* is the list of unconnected unknowns.

The unknown $x[j]$ is called the *owner* of column j, and $y[i]$ is called the *owner* of row i. The representation of an unknown includes the Boolean field *ownsRow* and the integer field *index*, which are maintained so that for any row owner $y[i]$, $y[i].ownsRow$ is **true** and $y[i].index = i$, and for any column owner $x[j]$, $x[j].ownsRow$ is **false** and $x[j].index = j$.

Recall that we are obliged to implement the procedure *UnknownForEnode*, specified in Section 4.4:

```
proc UnknownForEnode(e : E-node) : Unknown ≡
  if e.Simplex_Unknown ≠ nil then
    return e.Simplex_Unknown
  end;
  m := m + 1;
  x[m] := new(Unknown);
  Push ("Deallocate", x[m]) onto the undo stack (§3.1);
```

```
        x[m].enode := e
        x[m].ownsRow := false;
        x[m].index := m;
        for i := 1 to n do
           a[i, m] := 0
        end;
        e.Simplex_Unknown := x[m];
        return x[m]
     end
```

The assert procedures take formal affine sums of unknowns as arguments, but the hearts of their implementations operate on individual unknowns. We therefore introduce a procedure that converts an affine sum into an equivalent unknown, more precisely:

> **proc** *UnknownForFAS*(*fas* : formal affine sum of unknowns) : *Unknown*;
> /* Allocates and returns a new row owner constrained to be equal to *fas*. */

The implementation is a straightforward manipulation of the tableau data structure:

```
     proc UnknownForFAS(fas : formal affine sum of unknowns) : Unknown ≡
        n := n + 1;
        y[n] := new(Unknown);
        Push ("Deallocate", y[n]) on the undo stack (§3.1);
        y[n].ownsRow := true;
        y[n].index := n;
        let fas be the formal affine sum k₀ + k₁ × v₁ + · · · + kₚ × vₚ in
           a[n, 0] := k₀;
           for j := 1 to m do
              a[n, j] := 0
           end;
           for i := 1 to p do
              if vᵢ.ownsRow then
                 for j := 0 to m do
                    a[n, j] := a[n, j] + kᵢ × a[vᵢ.index, j]
                 end
              else if vᵢ.index > dcol then
                 a[n, vᵢ.index] := a[n, vᵢ.index] + kᵢ
              // else vᵢ owns a dead column and is constrained to be 0
              end
           end
        end;
        return y[n]
     end
```

8.3. TESTING CONSISTENCY. The *sample point* of the tableau is the point that assigns 0 to each column owner and that assigns $a[i, 0]$ to each row owner $y[i]$. Obviously, the sample point satisfies all the row constraints and dead column constraints. If it also satisfies all the sign constraints, then the tableau is said to be *feasible*. Our algorithm maintains feasibility of the tableau at all times.

A row owner $y[i]$ in a feasible tableau is said to be *manifestly maximized* if every non-zero entry $a[i, j]$, for $j > dcol$, is negative and lies in a column whose owner $x[j]$ is restricted. It is easy to see that in this case $y[i]$'s maximum value over *TableauSoln* is its *sample value* (i.e., its value at the sample point) $a[i, 0]$.

A live column owner $x[j]$ in a feasible tableau is said to be *manifestly unbounded* if every negative entry $a[i, j]$ in its column is in a row owned by an unrestricted unknown. It is easy to see that in this case *TableauSoln* includes points that assign arbitrarily large values to $x[j]$.

The *pivot* operation modifies the tableau by exchanging some row owner $y[j]$ with some column owner $x[i]$ while (semantically) preserving the tableau constraints. It is clear that, if $a[i, j] \neq 0$, this can be done by solving the $y[j]$ row constraint for $x[i]$ and using the result to eliminate $x[i]$ from the other row constraints. We leave it to the reader to check that the net effect of pivoting is to transform

	Pivot Column	Any Other Column
Pivot Row	a	b
Any Other Row	c	d

into

	Pivot Column	Any Other Column
Pivot Row	$1/a$	$-b/a$
Any Other Row	c/a	$d - bc/a$

We will go no further into the implementation of *Pivot*. The implementation in Nelson's thesis used the sparse matrix data structure described in Section 2.2.6 of Knuth's *Fundamental Algorithms* [Knuth 1968]. Using a sparse representation is probably a good idea, but in fact Simplify doesn't do so: it uses a dynamically allocated two-dimensional sequential array of rational numbers.

We impose upon the caller of $Pivot(i, j)$ the requirement that j be the index of a live column and that (i, j) is such that feasibility is preserved.

The pivot operation preserves not only *Asserted* but also *DColSoln*, *RowSoln*, and *SignSoln* individually.

An important observation concerns the effect of the pivot operation on the sample point. Let u be the owner of the pivot column. Since all other column owners have sample value 0 both before and after the pivot operation, while u is 0 before but not necessarily after the pivot operation, it follows that the sample point moves along the line determined by varying u while leaving all other column owners zero and setting each row owner to the unique value that satisfies its row constraint. We call this line the *line of variation* of u.

Here is the key fact upon which the Simplex algorithm is based:

> If u is any unknown of a feasible tableau, then there exists a sequence of pivots such that each pivot preserves feasibility, no pivot lowers the sample value of u, and the net effect of the entire sequence is to produce a tableau in which u is either manifestly maximized or manifestly unbounded. We call such a sequence of pivots a *rising sequence* for u.

Any of several simple rules will generate an appropriate sequence of pivots; no backtracking is required. The simplest pivot selection rule, described by George Dantzig in his original description of the Simplex Algorithm, is essentially to choose any pivot that increases (or at least does not decrease) the sample value of u while preserving feasibility. This is the rule that Simplify uses and we describe it in detail below. The reason we use it is that it is simple. A disadvantage is that it is not guaranteed to increase the sample value of u, and in fact it has been known for

many years that in the worst case this rule can lead to an infinite loop in which the Simplex algorithm pivots indefinitely. Unfortunately, the "lexicographic pivot selection rule", which prevents infinite looping, does not prevent exponentially many pivots in the worst case, which would be indistinguishable in practice from infinite looping. We therefore choose the simple rule, and put our faith in the overwhelming empirical evidence that neither infinite looping nor exponentially long rising sequences have any appreciable chance of occurring in practice.

Using this fact, and temporarily ignoring the details of how the pivots are chosen as well as the issue of propagating equalities, we can now describe an implementation of *AssertGE(fas)* in a nutshell:

> Create a row owner u whose row constraint is equivalent, given the existing constraints, to $u = fas$, and then pivot the tableau until either (1) u has a non-negative sample value or (2) u is manifestly maximized with a negative sample value. In case (1), restricting u leaves the tableau feasible and expresses the constraint $fas \geq 0$, so restrict u. In case (2), the inequality $fas \geq 0$ is inconsistent with the existing constraints, so set *refuted*.

Returning to the key fact, and the way in which the program exploits it, each pivot of the rising sequence for an unknown u is located by the routine *FindPivot*. The unknown u must be a row owner, and the argument to *FindPivot* is the index of its row. More precisely, given a row index i, *FindPivot(i)* returns a pair (h, j) for which one of the following three cases holds:

—$i \neq h$ and pivoting on (h, j) is the next pivot in a rising sequence for $y[i]$,

—$y[i]$ is manifestly maximized and $(h, j) = (-1, -1)$, or

—$i = h$ and pivoting on (h, j) produces a tableau in which $x[j]$ (formerly $y[i]$) is manifestly unbounded.

Using *FindPivot* (and *Pivot*), it is straightforward to determine the sign of the maximum of any given unknown. This task is performed by the routine *SgnOfMax*, whose specification is

```
proc SgnOfMax(u : Unknown) : integer ≡
/* Requires that u own a row or a live column. Returns −1, 0, or +1 according as the
maximum value of u over the solution set of the tableau is negative, zero, or positive. (An
unbounded unknown is considered to have a positive maximum.) In case 0 is returned, the
tableau is left in a state where u is manifestly maximized at zero. In case 1 is returned, the
tableau is left in a state where u is a row owner with positive sample value or is a manifestly
unbounded column owner. */
```

and whose implementation is

```
proc SgnOfMax(u : Unknown) : integer ≡
    if u owns a manifestly unbounded column then
        return 1
    end;
    ToRow(u); // pivots u to be a row owner, as described later
    while a[u.index, 0] ≤ 0 do
        (j, k) := FindPivot(u.index);
        if j = u.index then
            Pivot(j, k);
            // u is manifestly unbounded.
            return 1
        else if (j, k) = (−1, −1) then
```

```
            // u is manifestly maximized.
            return sign(a[u.index, 0])
        else
            Pivot(j, k)
        end
    end;
    return 1
end
```

We now turn to the implementation of *FindPivot*. *FindPivot*(i) chooses the pivot column first and then the pivot row.

Whatever column j is chosen, that column's owner $x[j]$ has current sample value zero, and when pivoted into a row, it will in general then have non-zero sample value. Since all other column owners will continue to be column owners and have sample value zero, the row constraint for row i implies that the change to the sample value of the row owner $y[i]$ caused by the pivot will be $a[i, j]$ times the change in the sample value of the pivot column owner $x[j]$ (which is also the post-sample-value of that unknown). In order for the pivot to have any chance of increasing the sample value of $y[i]$, it is therefore necessary that $a[i, j] \neq 0$. Furthermore, if $x[j]$ is restricted, then we can only use pivots that increase its sample value. In this case, the sample value of $y[i]$ will increase only if $a[i, j] > 0$. In summary, we choose any live pivot column with a positive entry in row i or with a negative entry in row i and an unrestricted column owner. Notice that if there is no such column, then row i's owner is manifestly maximized, and *FindPivot* can return $(-1, -1)$.

Having chosen the pivot column j, we must now choose the pivot row. To choose the pivot row, we begin by recalling that each pivot in column j moves the sample point along the line of variation of $x[j]$. Each restricted row owner $y[h]$ whose entry $a[h, j]$ in the pivot column has sign opposite to $a[i, j]$ imposes a limit on how far the sample point may be moved along the line of variation of $x[j]$ in the direction that increases $y[i]$. If there are no such restricted row owners, then the solution set contains an infinite ray in which the pivot column owner takes on values arbitrarily far from zero in the desired direction, and therefore $y[i]$ takes on arbitrarily large values. In fact, as the reader can easily show, pivoting on (i, j) will in this case move $y[i]$ to be a manifestly unbounded column owner. On the other hand, if one or more rows do impose restrictions on the distance by which the pivot column owner can be moved, then we choose as the pivot row any row h that imposes the strictest restriction. Pivoting on (h, j) makes $y[h]$ into a column owner with sample value 0 and therefore obeys the strictest restriction, and hence all the restrictions.

```
proc FindPivot(i : integer) : (integer × integer) ≡
    var j, sgn in
        if there exists a k > dcol such that ¬ x[k].restricted and a[i, k] ≠ 0 then
            Choose such a k;
            (j, sgn) := (k, a[i, k])
        else if there exists a k > dcol such that
            x[k].restricted and a[i, k] > 0 then
            Choose such a k;
            (j, sgn) := (k, +1)
        else
            return (−1, −1)
        end;
        // Column j is the pivot column, and the sign of sgn is
```

```
        // the direction in which the pivot should move x[j].
        var champ := i, score := ∞ in
            for each row h such that h ≠ i and y[h].restricted do
                if sgn × a[h, j] < 0 ∧ |a[h, 0]/a[h, j]| < score then
                    score := |a[h, 0]/a[h, j]|;
                    champ := h
                end
            end;
            return (champ, j)
        end
    end
end
```

We next give the specification and implementation of *ToRow*, which is similar to the second half of *FindPivot*.

```
proc ToRow(u : Unknown) ≡
/*  Requires that u not own a dead or manifestly unbounded column. A no-op if u already
owns a row. If u owns a column, pivots u into a row.  */
```

```
proc ToRow(u : Unknown) ≡
    if u.ownsRow then
        return
    end;
    var j := u.index, champ, score := ∞ in
        for each row h such that y[h].restricted ∧ a[h, j] < 0 do
            if −a[h, 0]/a[h, j] < score then
                score := −a[h, 0]/a[h, j];
                champ := h
            end
        end;
        Pivot(champ, j)
    end
end
```

If u owns a column, then *ToRow* must pivot u into a row, while preserving feasibility of the tableau. Recall that any pivot in u's column leaves the sample point somewhere on the line of variation of u. If u were manifestly unbounded, then the entire portion of the line of variation of u for which $u > 0$ would lie inside the solution set of *TableauSoln*. But it is a precondition of *ToRow* that u not be manifestly unbounded. That is, there is at least one restricted row owner whose row has a negative entry in u's column. The sign constraint of each such row owner $y[h]$ imposes a bound on how far the sample point can be moved along the line of variation of u in the direction of increasing u while still remaining in the solution set. Specifically, since $a[h, u.index]$ is negative, $y[h]$ decreases as u increases on the line of variation of u, and u therefore must not be increased beyond the point at which $y[h] = 0$. *ToRow* chooses a restricted row owner whose sign constraint imposes the strictest bound on the increasing motion of u along the line of variation of u and moves the sample point to that bound by pivoting u's column with that row, thereby leaving the chosen row owner as a column owner with sample value 0.

We are now ready to give the implementation for *AssertGE*:

```
proc AssertGE(fas : formal affine sum of connected unknowns) ≡
    var u := UnknownForFAS(fas) in
```

```
         var sgmx := SgnOfMax(u) in
           if sgmx < 0 then
              refuted := true;
              return
           else
              u.restricted := true;
              Push ("Unrestrict", u) onto the undo stack (§3.1);
              if sgmx = 0 then
                 CloseRow(u.index)
              end;
              return
           end
        end
     end
  end
```

The fact that *AssertGE* propagates enough equalities in the case that *SgnOfMax* returns 0 is a consequence of the specification for *CloseRow*:

proc *CloseRow*(*i* : **integer**);
/* Requires that *y*[*i*] be a restricted unknown *u* that is manifestly maximized at zero. Establishes PROPAGATION CORRECTNESS. May modify *dcol* and thus modify the dead column constraints, but preserves the solution set of the tableau. */

The fact that *AssertGE* is correct to propagate no equalities in the case that *SgnOfMax* returns 1 is a consequence of the following lemma:

LEMMA 1.   *Suppose that an unknown u has a positive value at some point in TableauSoln. Then adding the sign constraint $u \geq 0$ does not increase the set of affine equalities that hold over the solution set.*

PROOF.   Let f be an affine function of the unknowns that is zero over the intersection of *TableauSoln* with the half-space $u \geq 0$. We will show that $f(P) = 0$ for all $P \in TableauSoln$. By the hypothesis, $f(P) = 0$ if $u(P) \geq 0$. It remains to consider the case that $u(P) < 0$.

By the assumption of the lemma, there is a point $R \in TableauSoln$ such that $u(R) > 0$. Since u is negative at P and positive at R, there is a point $Q \neq R$ on the line segment PR such that $u(Q) = 0$. Note that $Q \in TableauSoln$, since *TableauSoln* is convex. Since $u(R) > 0$ and $u(Q) = 0$, both R and Q lie in the half-space $u \geq 0$. Since $f = 0$ holds over the intersection of *TableauSoln* with this half-space, it follows that $f(R)$ and $f(Q)$ are both zero. Since the affine function f is zero at two distinct points, Q and R, on the line PR, it must be zero on the entire line, including at P.   □

It would be possible to implement *AssertZ* using two calls to *AssertGE*, but we will show a more efficient implementation below.

8.3.1. *Simplex Redundancy Filtering*.   Simplify's *AssertGE* incorporates an optimization *Simplex redundancy filtering* not shown in the pseudo-code above. It sometimes happens that an assertion is trivially redundant, like the second assertion of *AssertLit*($n \geq 0$); *AssertLit*($n + 1 \geq 0$). *AssertGE* filters out such redundant assertions. The technique is to examine the row owned by an unknown that is about to be restricted. If every nonzero entry in the row is positive and lies either in the constant column or in a column with a restricted owner, then the assertion for which the row was created is redundant, so instead of restricting the row owner and testing for consistency, *AssertGE* simply deletes the row.

8.4. PROPAGATING EQUALITIES.   It remains to implement *CloseRow*, whose job is to handle the case where some unknown *u* has been constrained to be at least zero in a tableau that already implies that *u* is at most zero. In this case, the dimensionality of the solution set is potentially reduced by the assertion, and it is potentially necessary to propagate equalities as required by the equality-sharing (§4.1) protocol.

Two unknowns are said to be *manifestly equal* if either (1) both own rows, and those rows are identical, excluding entries in dead columns; (2) one is a row owner $y[i]$, the other is a column owner $x[j]$, and row i, excluding entries in dead columns, contains only a single nonzero element $a[i, j] = 1$; (3) both own dead columns; or (4) one owns a dead column and the other owns a row whose entries, excluding those in dead columns, are all zero. It is easy to see that if *u* and *v* are manifestly equal, the equality $u = v$ holds over *TableauSoln* (and, in fact, over *HPlaneSoln*).

Detecting equalities implied by the hyperplane constraints (§8.2) alone is straightforward, as shown by the following lemma:

LEMMA 2.   *An equality between two unknowns of the tableau is implied by the hyperplane constraints iff the unknowns are manifestly equal.*

PROOF.   The hyperplane constraints imply that each unknown is equal to a certain formal affine sum of the owners of the live columns, as follows:

For each row owner, $y[i]$: $y[i] = a[i, 0] + \Sigma_{j:dcol < j \leq m} a[i, j] \times x[j]$.

For each live column owner, $x[k]$: $x[k] = 0 + \Sigma_{j:dcol < j \leq m} \delta_{jk} \times x[j]$, where δ_{ij} is 1 if $i = j$ and 0 otherwise.

For each dead column owner, $x[k]$: $x[k] = 0 + \Sigma_{j:dcol < j \leq m} 0 \times x[j]$.

Since, in satisfying the hyperplane constraints, the values of the live column owners can be assigned arbitrarily, two unknowns are equal over *HPlaneSoln* iff the corresponding formal affine sums are identical in all coefficients. It is straightforward to check that coefficients are identical precisely when the unknowns are manifestly equal.   □

The next lemma gives a simple condition under which the problem of finding equalities that hold over *TableauSoln* reduces to that of finding equalities that hold over *HPlaneSoln*:

LEMMA 3.   *Suppose that for each restricted unknown u, either u is manifestly equal to Zero or TableauSoln includes a point at which u is strictly positive. Then for any affine function f of the unknowns,*

$$f = 0 \text{ over TableauSoln} \iff f = 0 \text{ over HPlaneSoln}.$$

PROOF.   The "⇐" part is a consequence of the fact that *TableauSoln* ⊆ *HPlaneSoln*.

The "⇒" part follows by induction on the number of restricted unknowns not manifestly equal to *Zero*. The base case follows from the fact that if all restricted unknowns are manifestly equal to *Zero*, then *HPlaneSoln* coincides with *TableauSoln*. The induction step is simply an instance of Lemma 1.   □

When each restricted unknown either is positive somewhere in *TableauSoln* or is manifestly equal to *Zero*, we say that the tableau is *minimal*. (This name comes from the fact that a minimal tableau represents *Asserted* using the smallest possible number of live columns, namely the dimension of the solution set.) An immediate consequence of Lemmas 2 and 3 and the definition of *Asserted* is:

LEMMA 4. *Suppose that the tableau is consistent and minimal. Then an equality between two connected unknowns is implied by Asserted iff it is manifest.*

The procedure *CloseRow* uses *SgnOfMax* to find any restricted unknowns that cannot assume positive values in *TableauSoln*, and which therefore must be equal to zero over *TableauSoln*. When it finds such unknowns, it makes their equality to *Zero* manifest, using the procedure *KillCol*, which is specified as follows:

proc *KillCol*(j : **integer**)
/* Requires that column j be owned by an unknown $u = x[j]$ having value 0 over *TableauSoln*, and that all manifest equalities of connected unknowns are implied by *Propagated*. Adds the dead column constraint $u = 0$ (undoably), leaving *Asserted* unchanged, and propagates equalities as necessary to ensure that all manifest equalities of connected unknowns are still implied by *Propagated*. */

As *CloseRow* applies *KillCol* to more and more columns the tableau will eventually become minimal. At this point, Lemma 4 and the postcondition of the last call to *KillCol* will together imply PROPAGATION CORRECTNESS.

```
proc KillCol( j : integer) ≡
    Push "ReviveCol" onto the undo stack (§3.1);
    swap column j with column dcol + 1;
    dcol := dcol + 1;
    propagate an equality between x[dcol] and Zero;
    for each i such that y[i] is connected and a[i, dcol] ≠ 0 do
        if y[i] is manifestly equal to some connected unknown u then
            propagate an equality between y[i] and some such u
        end
    end
end

proc CloseRow(i) ≡
    for each j > dcol such that a[i, j] ≠ 0 do
        // a[i, j] < 0
        KillCol( j)
    end;
    var R := the set of all restricted unknowns with
        sample value 0 but not manifestly equal to Zero in
        while R ≠ { } do
            var u := any element of R in
                delete u from R;
                if SgnOfMax(u) = 0 then
                    // u owns a row and every non-zero entry in that row is in a
                    // column whose owner has value 0 over TableauSoln.
                    for each j > dcol such that a[u.index, j] ≠ 0 do
                        // a[u.index, j] < 0
                        KillCol( j)
                    end
                    // u is now manifestly equal to Zero
                else
                    // SgnOfMax(u) = +1, so skip
                end;
                delete from R all elements that are manifestly equal to Zero or
                    have positive sample values
            end
        end
    end
end
```

The reader might hope that the first three lines of this procedure would be enough and that the rest of it is superfluous, but the following example shows otherwise:

	1	$^\nabla p^\geq$	q	r
Zero	0	0	0	0
$a^\geq$	0	-1	0	0
$b^\geq$	0	1	-1	1
$c^\geq$	0	1	1	$-1.$

In the tableau above, the unknown a is manifestly maximized at zero. When *CloseRow* is applied to a's row, the first three lines kill the column owned by p. This leaves a nonminimal tableau, since the tableau then implies $b = c = 0$ and that $q = -r$. The rest of *CloseRow* will restore minimality by pivoting the tableau to make either b or c a column owner and killing the column.

As a historical note, the description in Nelson's thesis [Nelson 1979] erroneously implied that *CloseRow* could be implemented with the first three lines alone. This bug was also present in our first implementation of Simplify and went unnoticed for a considerable time, but eventually an ESC example provoked the bug and we diagnosed and fixed it. We note here that the fix we implemented years ago is different from the implementation above, so the discrepancy between paper and code is greater in this section than in the rest of our description of the Simplex module.

As remarked above, *AssertZ* could be implemented with two calls to *AssertGE*. But if it were, the second of the two calls would always lead to a call to *CloseRow*, which would incur the cost of reminimizing the tableau by retesting the restricted unknowns. This work is not always necessary, as shown by the following lemma:

LEMMA 5. *If an unknown u takes on both positive and negative values within TableauSoln, and a restricted unknown v is positive somewhere in TableauSoln, then $(v > 0 \land u = 0)$ must hold somewhere in TableauSoln.*

PROOF. Let P be a point in *TableauSoln* such that $v(P) > 0$. If $u(P) = 0$, we are done. Otherwise, let Q be a point in *TableauSoln* such that $u(Q)$ has sign opposite to $u(P)$. Since *TableauSoln* is convex, line segment PQ lies entirely within *TableauSoln*. Let R be the point on PQ such that $u(R) = 0$. Since v is restricted, $v(Q) \geq 0$. Since $v(Q) \geq 0$, $v(P) > 0$, $R \in PQ$, and $R \neq Q$, it follows that $v(R) > 0$. □

We will provide an improved implementation of *AssertZ* that takes advantage of Lemma 5. The idea is to use two calls to *SgnOfMax* to determine the signs of the maximum and minimum of the argument. If these are positive and negative respectively, then by Lemma 5, we can perform the assertion by killing a single column. We will need to modify *SgnOfMax* to record in the globals (*iLast*, *jLast*) the row and column of the last pivot performed (if any). Since *SgnOfMax(u)* never performs any pivots after u acquires a positive sample value, and since pivoting is its own inverse, it follows that this modified *SgnOfMax* satisfies the additional postcondition:

> If u's initial sample value is at most zero, and u ends owning a row and 1 is returned, then the tableau is left in a state such that pivoting at (*iLast*, *jLast*) would preserve feasibility and make u's sample value at most zero.

Using the modified *SgnOfMax*, we implement *AssertZ* as follows:

```
proc AssertZ(fas : formal affine sum of connected unknowns) ≡
    var u := UnknownForFAS(fas) in
        if u is manifestly equal to Zero then
            return
        end;
        var sgmx := SgnOfMax(u) in
            if sgmx < 0 then
                refuted := true;
                return
            else if sgmx = 0 then
                CloseRow(u.index);
                return
            else
                // Sign of max of fas is positive.
                if u.ownsRow then
                    for j := 0 to m do
                        a[u.index, j] := −a[u.index, j]
                    end;
                else
                    for i := 1 to n do
                        a[i, u.index] := −a[i, u.index]
                    end
                end;
                // u now represents −fas.
                sgmx := SgnOfMax(u) in
                if sgmx < 0 then
                    refuted := true;
                    return
                else if sgmx = 0 then
                    CloseRow(u.index);
                    return
                else
                    // Sign of max of −fas is also positive, hence Lemma 5 applies.
                    if u.ownsRow then
                        Pivot(u.index, jLast);
                    end;
                    KillCol(u.index);
                    return
                end
            end
        end
    end
end
```

The main advantage of this code for *AssertZ* over the two calls to *AssertGE* occurs in the case where both calls to *SgnOfMax* return 1. In this case, the unknown *u* takes on both positive and negative values over the initial *TableauSoln*, so we know by Lemma 5 that if any restricted unknown *v* is positive for some point in the initial *TableauSoln*, *v* is also positive at some point that remains in *TableauSoln* after the call to *KillCol* adds the dead column constraint *u* = 0. Thus, the minimality of the tableau is guaranteed to be preserved without any need for the loop over the restricted unknowns that would be caused by the second of the two *AssertGE* assertions.

We must also show that feasibility of the tableau is preserved in the case where both calls to *SgnOfMax* return 1. In this case, the first call to *SgnOfMax* leaves *u*

with sample value at least 0. After the entries in u's row or column are negated, u then has sample value at most 0. If the second call to *SgnOfMax* then leaves u as a row owner, the postconditions of *SgnOfMax* guarantee that $u \geq 0$ at the sample point, and that pivoting at $(iLast, jLast)$ would move the sample point to a point on the line of variation (§8.3) of $x[jLast]$ where $u \leq 0$ while preserving feasibility. Instead, we pivot at $(u.index, jLast)$, which moves the sample point to the point along the line of variation of $x[jLast]$ to the point where $u = 0$. By the convexity of *TableauSoln*, this pivot also preserves feasibility.

A final fine point: The story we have told so far would require a tableau row for each distinct numeric constant appearing in the conjecture. In fact, Simplify incorporates an optimization that often avoids creating unknowns for numeric constants. A price of this optimization is that it becomes necessary to propagate equalities not only between unknowns but also between an unknown and a numeric constant. For example, when Simplify interns (§4.5) and asserts a literal containing the term $f(6)$, it creates an E-node for the term 6 but does not connect it to a Simplex unknown. On the other hand, if the Simplex tableau later were to imply $u = 6$, for some connected unknown u, Simplify would then ensure that an equality is propagated between $u.enode$ and the E-node for 6. The changes to *CloseRow* required to detect unknowns that have become "manifestly constant" are straightforward.

8.5. Undoing Tableau Operations.   The algorithms we have presented push undo records when they modify *SignSoln*, *DColSoln*, *RowSoln*, or the set of connected unknowns. In this section, we describe how these undo records are processed by *Pop*. Since the representation of *Asserted* is given as a function of *SignSoln*, *DColSoln*, *RowSoln*, it follows from the correctness of these bits of undoing code that *Pop* meets its specification of restoring *Asserted*.

Notice that we do not push an undo record for *Pivot*. As a consequence *Pop* may not restore the initial permutation of the row and column owners. This doesn't matter as long as the semantics of *SignSoln*, *DColSoln*, *RowSoln* are restored.

To process an undo record of the form (`"Unrestrict"`, u):

$\quad u.restricted := false.$

To process an undo record of the form `"ReviveCol"`:

$\quad dcol := dcol - 1.$

To process an undo record of the form (`"Deallocate"`, u):

```
if u.ownsRow then
    copy row n to row u.index;
    y[u.index] := y[n];
    n := n − 1
else if u's column is identically 0 then
    copy column m to column u.index;
    x[u.index] := x[m];
    m := m − 1
else
    perform some pivot in u's column that preserves feasibility;
    copy row n to row u.index;
    y[u.index] := y[n];
    n := n − 1
end;
```

```
if u.enode ≠ nil then
    u.enode.Simplex_Unknown := nil;
    u.enode := nil
end
```

In arguing the correctness of this undo action, we observe that there are two cases in which an unknown is allocated: *UnknownForEnode* allocates a completely un-constrained unknown and connects it to an E-node, and *UnknownForFAS* allocates a constrained unknown and leaves it unconnected. If the unknown u to be deallocated was allocated by *UnknownForEnode*, then at entry to the action above the tableau will have been restored to a state where u is again completely unconstrained. In this case, u must own an identically zero column. All that is required to undo the forward action is to delete the vacuous column, which can be done by overwriting it with column m and decrementing m. If u was allocated by *UnknownForFAS*, then u will be constrained by *RowSoln*. If u happens to a row owner, we need only delete its row. If u owns a column, we cannot simply delete the column (since that would change the projection of *RowSoln* onto the remaining unknowns), so we must pivot u to make it a row owner. The argument that it is always possible to do this while preserving feasibility is similar to those for *ToRow* and for *FindPivot*, and we leave the details to the reader.

8.6. INTEGER HEURISTICS.   As is well known, the satisfiability problem for linear inequalities over the rationals is solved in polynomial time by various ellipsoid methods and solved rapidly in practice by the Simplex method, but the same problem over the integers is *NP*-complete: Propositional unknowns can be constructed from integer unknowns by adding constraints like $0 \le b \wedge b \le 1$, after which $\neg b$ is encoded $1 - b$, and $b \vee c \vee d$ is encoded $b + c + d > 0$. Thus, 3SAT is encoded.

For integer linear inequalities that arise from such an encoding, it seems extremely likely (as likely as $P \ne NP$) that the only way to solve the satisfiability problem will be by some kind of backtracking search. A fundamental assumption of Simplify is that most of the arithmetic satisfiability problems arising in program checking don't resemble such encoded SAT problems and don't require such a backtracking search.

In designing Simplify, we might have introduced a built-in monadic predicate characterizing the integers. Had we done so, a conjecture could mention both integer and rational unknowns. But rational unknowns did not seem useful in our program-checking applications. (We did not aspire to formulate a theory of floating-point numbers.) So Simplify treats all arithmetic arguments and results as integers.

The Simplex module's satisfiability procedure is incomplete for integer linear arithmetic. However, by combining the complete decision procedure for rational linear arithmetic described above with three heuristics for integer arithmetic, we have obtained satisfactory results for our applications.

The first heuristic, the *negated inequality heuristic*, is that to assert a literal of the form $\neg a \le b$, Simplify performs the call *AssertGE*$(a - b - 1)$.

We found this heuristic alone to be satisfactory for a number of months. It is surprisingly powerful. For example, it allows proving the following formula:

$$i \le n \wedge f(i) \ne f(n) \Rightarrow i \le n - 1.$$

The reason this formula is proved is that the only possible counterexample (§6.1)

$$i \leq n, \quad f(i) \neq f(n), \quad \neg i \leq n - 1,$$

and the heuristically transformed third literal $i - (n - 1) - 1 \geq 0$ combines with the first literal to propagate the equality $i = n$, contradicting the second literal.

Eventually, we encountered examples for which the negated inequality heuristic alone was insufficient. For example:

$$2 \leq i \wedge f(i) \neq f(2) \wedge f(i) \neq f(3) \Rightarrow 4 \leq i.$$

We therefore added a second integer heuristic, the *Tighten Bounds* proof tactic. If, for some term t that is not an application of $+$, $-$, or $\times$, the Simplex tableau contains upper and lower bounds on t: $L \leq t$ and $t \leq U$, for integer constants L and U, Simplify will try to refute the possibility $t = L$ (not try so hard as to do a case split, but Simplify will do matching on both unit and nonunit matching rules (§5.2) in the attempted refutation), and if it is successful, it will then strengthen the lower bound from L to $L + 1$ and continue. It would be perfectly sound to apply the tactic to terms that were applications of $+$, $-$, or $\times$, but for efficiency's sake we exclude them.

The Tighten Bounds tactic is given low priority. Initially, it had the lowest priority of all proof tactics: it was only tried as a last resort before printing a counterexample. However, we later encountered conjectures whose proof required the use of Tighten Bounds many times. We therefore introduced a Boolean recording whether the tactic had recently been useful. When this Boolean is true, the priority of the tactic is raised to just higher than case splitting. The Boolean is set to true whenever the Tighten Bounds tactic is successful, and it is reset to false when Simplify backtracks from a case split on a non-goal clause and the previous case split on the current path (§3.1) was on a goal (§3.5) clause. That is, the boolean is reset whenever clause scores are renormalized (as described in Section 3.6).

Simplify's third integer arithmetic heuristic is the *manifest constant* heuristic. If all live column entries in some row of the tableau are zero, then the tableau implies that the row owner is equal to the entry in its constant column entry. If the constant column entry in such a row is not an integer, then the Simplex module detects a contradiction and sets *refuted*.

8.7. OVERFLOW.   Simplify implements the rational numbers in the simplex tableau as ratios of 32-bit or 64-bit integers, depending on the platform. By default, Simplify does no overflow checking. If an overflow does occur it can produce a crash or an incorrect result. Several of the bug reports that we have received have been traced to such overflows. Simplify accepts a switch that causes it to check each tableau operation for overflow and halt with an error message if an overflow occurs. When ESC/Java is applied to its own front end, none of the proofs of the 2331 verification conditions overflows on a 64-bit machine; one of them overflows on a 32-bit machine. Enabling overflow checking increases total proof time by about four percent on the average.

9. *Performance Measurements*

In this section, we report on the performance of Simplify and on the performance effects of heuristics described in the rest of the article.

All performance figures in this section are from runs on a 500-MHz Compaq Alpha (EV6) with 5-GB main memory. The most memory used by Simplify on the "front-end test suite", defined below, seems to be 100 MB. The machine has three processors, but Simplify is single-threaded. Simplify is coded in Modula-3 with all bounds checking and null-dereference checks enabled. Simplify relies on the Modula-3 garbage collector for storage management.

Our goal was to achieve performance sufficient to make the extended static checker useful. With the released version of Simplify, ESC/Java is able to check the 44794 thousand lines of Java source in its own front end (comprising 2331 routines and 29431 proof obligations) in 91 minutes. This is much faster than the code could be checked by a human design review, so we feel we have succeeded. We have no doubt that further performance improvements (even dramatic ones) are possible, but we stopped working on performance when Simplify became fast enough for ESC.

We used two test suites to produce the performance data in this section. The first suite, which we call the "small test suite" consists of five verification conditions (§1) generated by ESC/Modula-3 (`addhi`, `cat`, `fastclose`, `frd-seek`, `simplex`), eleven verification conditions generated by ESC-Java:

```
toString
isCharType
visitTryCatchStmt
binaryNumericPromotion
Parse
getRootInterface
checkTypeDeclOfSig
checkTypeDeclElem
main
getNextPragma
scanNumber
```

and two artificial tests (`domino6x4x` and `domino6x6x`) that reduce the well-known problem of tiling a mutilated checkerboard with 1-by-2 dominos into test cases that exercise case splitting and the E-graph. The second suite, which we call the "front-end test suite" consists of the 2331 verification conditions for the routines of ESC/Java's front end. Both test suites contain valid conjectures only. These tests are available on the web [Detlefs et al. 2003a].

Much of our performance work aimed not so much at improving the average case as at steering clear of the worst case. Over the course of the ESC project, Simplify would occasionally "go off the deep end" by diving into a case analysis whose time to completion was wildly longer than any user would ever wait. When this happened, we would analyze the problem and the computation into which Simplify stumbled, and design a change that would prevent the problem from occurring. The change might not improve average case behavior, but we took the view that going off the deep end on extended static checking of realistic programs was worse than simple slowness.

Of course, it could be that the next realistic program that ESC is applied to will send Simplify off the deep end in a way that we never encountered and never took precautions against. Therefore, when ESC uses Simplify to check a routine, it sets a timeout. If Simplify exceeds the timeout, ESC notifies the user and goes on to

benchmark	time	splits	incs	max
addhi	3.39	82	4	1
cat	10.25	253	6	2
fastclose	2.37	70	0	0
frd-seek	6.45	171	1	1
simplex	28.23	134	0	0
domino6x4x	2.23	1194	0	0
domino6x6x	47.71	20809	0	0
toString	1.75	0	0	0
isCharType	1.71	0	0	0
visitTryCatchStmt	2.98	3	0	0
binaryNumericPromotion	2.38	10	0	0
Parse	7.42	70	0	0
getRootInterface	8.34	20	0	0
checkTypeDeclOfSig	64.07	353	0	0
checkTypeDeclElem	88.36	349	0	0
main	5.98	18	0	0
getNextPragma	136.95	15365	234	1
scanNumber	21.77	2700	22	1

FIG. 4.   Baseline performance data for the small test suite.

check the next routine. The default time limit used by ESC/Java, and in the tests reported in this section, is 300 seconds.

Many of the optimizations that Simplify employs can be disabled, and Simplify has other options and configurable parameters. In Sections 9.2 through 9.13, we evaluate the performance effects of the most important of these options. We don't, of course, test every combination of options, but we have tried to test each option against the *baseline* behavior in which all options have their default values. Sections 9.14 through 9.17 discuss more general performance issues.

One of the verifications condition in the front end test suite requires approximately 400 seconds with Simplify's default options, and thus times out with our 300-second time limit. No other test in either suite times out.

Figure 4 gives the baseline performance data for the tests in the small suite. For each benchmark, the figure presents the benchmark name, the time in seconds for Simplify to prove the benchmark, the number of case splits performed, the number of times during the proof that the matching depth (§5.2.1) was incremented, and the maximum matching depth that was reached.

9.1. SIMPLIFY AND OTHER PROVERS.   As we go to press, Simplify is a rather old system. Several more recent systems that use decision procedures are substantially faster than Simplify on unquantified formulas. This was shown by a recent performance study performed by de Moura and Ruess [2004]. Even at its advanced age, Simplify seems to be a good choice for quantified formulas that require both matching and decision procedures.

9.2. PLUNGING.   As described in Section 4.6, Simplify uses the plunging heuristic, which performs a *Push-Assert-Pop* sequence to test the consistency of a literal with the current context, in an attempt to refine the clause containing the literal. Plunging is more complete than the E-graph tests (§4.6), but also more expensive. By default Simplify plunges on each literal of each clause produced by matching a non-unit matching rule (§5.2): untenable literals are deleted from the clause, and if any literal's negation is found untenable, the entire clause is deleted.

benchmark	time	(change)	splits	incs	max
addhi	6.14	(+81%)	1347	9	1
cat	>500.00	(>+4000%)	81107	32	2
fastclose	2.38	(+0%)	70	0	0
frd-seek	6.09	(−6%)	209	4	1
simplex	12.31	(−56%)	154	0	0
domino6x4x	2.12	(−5%)	1194	0	0
domino6x6x	47.44	(−1%)	20809	0	0
toString	1.75	(+0%)	0	0	0
isCharType	1.80	(+5%)	0	0	0
visitTryCatchStmt	3.10	(+4%)	3	0	0
binaryNumericPromotion	2.46	(+3%)	10	0	0
Parse	6.10	(−18%)	132	0	0
getRootInterface	8.37	(+0%)	20	0	0
checkTypeDeclOfSig	60.46	(−6%)	353	0	0
checkTypeDeclElem	79.67	(−10%)	349	0	0
main	6.13	(+3%)	39	3	1
getNextPragma	218.59	(+60%)	30118	234	1
scanNumber	18.87	(−13%)	3314	22	1

FIG. 5.   Performance data for the small test suite with plunging disabled.

Early in Simplify's history, we tried refining all clauses by plunging before doing each case-split. The performance of this strategy was so bad that we no longer even allow it to be enabled by a switch, and so cannot reproduce those results here.

We do have a switch that turns off plunging. Figure 5 compares, on the small test suite, the performance of Simplify with plunging disabled to the baseline. The figure has the same format as Figure 4 with an additional column that gives the percentage change in time from the baseline. The eighteen test cases of the small suite seem insufficient to support a firm conclusion. But, under the principle of steering clear of the worst case, the line for cat is a strong vote for plunging.

Figure 6 below illustrates the performance effect of plunging on each of the front end test suite. The figure contains one dot for each routine in the front end. The dot's x-coordinate is the time required to prove that routine's verification condition by the baseline Simplify and its y-coordinate is the time required by Simplify with plunging disabled. As the caption of the figure shows, plunging reduces the total time by about five percent. Moreover, in the upper right of the figure, all dots that are far from the diagonal are above the diagonal. Thus, plunging both improves the average case behavior and is favored by the principle of steering clear of the worst case.

The decision to do refinement by plunging still leaves open the question of how much effort should be expended during plunging in looking for a contradiction. If asserting the literal leads to a contradiction in one of Simplify's built-in theory modules, then the matter is settled. But if not, how many inference steps will be attempted before admitting that the literal seems consistent with the context? Simplify has several options, controlled by a switch. Each option calls *AssertLit*, then performs some set of tactics to quiescence, or until a contradiction is detected. Here's the list of options (each option includes all the tactics the preceding option).

0 Perform domain-specific decision procedures and propagate equalities.

1 Call (a nonplunging version of) *Refine*.

2 Match on unit rules (§5.2).

3 Match on non-unit rules (but do not plunge on their instances).
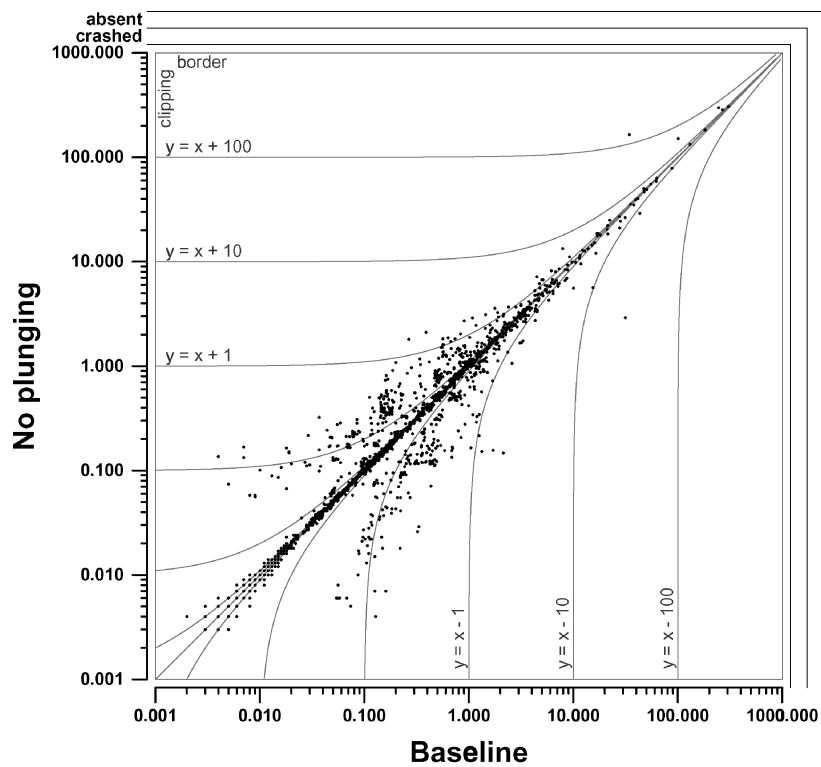
Checked 2331 routines in 4030 sec. (vs. 3841 sec. for baseline)
Includes 1 timeout.

Fig. 6.   Effect of disabling plunging on the front-end test suite.

Simplify's default is option 0, which is also the fastest of the options on the front-end test suite. However, option 0, option 1, and no plunging at all are all fairly close. Option 2 is about twenty percent worse, and option 3 is much worse, leading to many timeouts on the front-end test suite.

9.3. THE MOD-TIME OPTIMIZATION.   As described in Section 5.4.2, Simplify use the mod-time optimization, which records modification times in E-nodes in order to avoid fruitlessly searching for new matches in unchanged portions of the E-graph. Figure 7 shows the elapsed times with and without the mod-time optimization for Simplify on our small test suite.

For the two domino problems, there is no matching, and therefore the mod-time feature is all cost and no benefit. It is therefore no surprise that mod-time updating caused a slowdown on these examples. The magnitude of the slowdown is, at least to us, a bit of a surprise, and suggests that the mod-time updating code is not as tight as it could be. Despite this, on all the cases derived from program checking, the mod-time optimization was either an improvement (sometimes quite significant) or an insignificant slowdown.

Figure 8 shows the effect of disabling the mod-time optimization on the front-end test suite. Grey triangles are used instead of dots if the outcome of the verification is different in the two runs. In Figure 8, the two triangles in the upper right corner are cases where disabling the mod-time optimization caused timeouts. The hard-to-see

benchmark	time	(change)	splits	incs	max
addhi	3.52	(+4%)	80	4	1
cat	11.40	(+11%)	237	6	2
fastclose	2.73	(+15%)	70	0	0
frd-seek	7.95	(+23%)	171	1	1
simplex	43.61	(+54%)	134	0	0
domino6x4x	0.83	(−63%)	1194	0	0
domino6x6x	11.70	(−75%)	20809	0	0
toString	1.48	(−15%)	0	0	0
isCharType	1.60	(−6%)	0	0	0
visitTryCatchStmt	3.11	(+4%)	3	0	0
binaryNumericPromotion	2.25	(−5%)	10	0	0
Parse	8.77	(+18%)	63	0	0
getRootInterface	11.64	(+40%)	20	0	0
checkTypeDeclOfSig	73.60	(+15%)	353	0	0
checkTypeDeclElem	107.77	(+22%)	349	0	0
main	5.98	(+0%)	18	0	0
getNextPragma	>500.00	(>+200%)	21158	193	1
scanNumber	41.22	(+89%)	2135	10	1

FIG. 7.   Performance data for the small test suite with the mod-time optimization disabled.
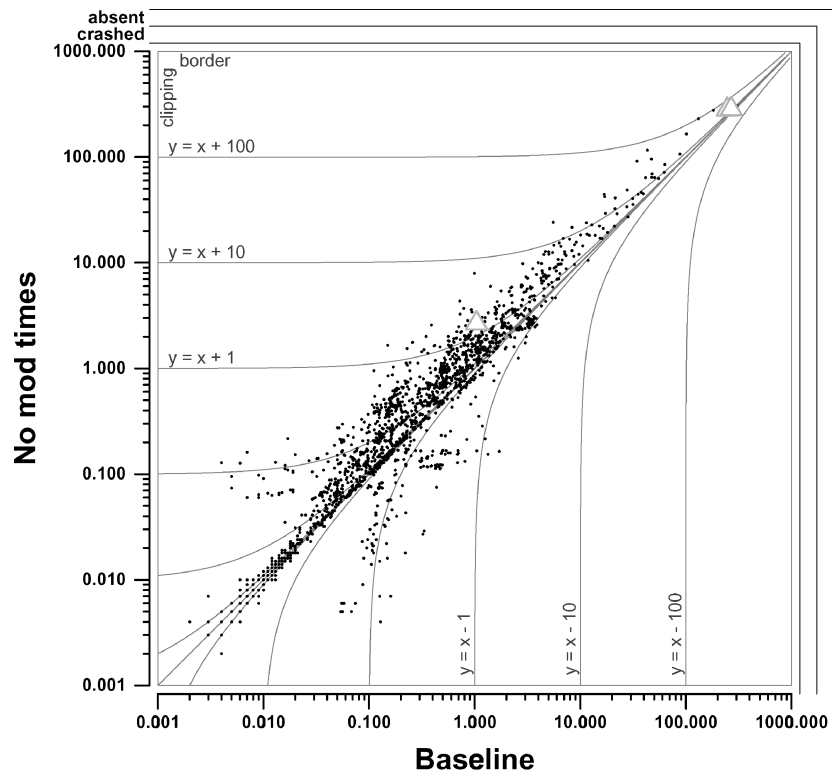
Checked 2331 routines in 5377 sec. (vs. 3841 sec. for baseline)
Includes 3 timeouts.

FIG. 8.   Effect of disabling the mod-time optimization on the front-end test suite.

benchmark	time	(change)	splits	incs	max
addhi	3.63	(+7%)	78	4	1
cat	12.73	(+24%)	253	6	2
fastclose	2.69	(+14%)	70	0	0
frd-seek	7.33	(+14%)	171	1	1
simplex	32.00	(+13%)	134	0	0
domino6x4x	2.05	(−8%)	1194	0	0
domino6x6x	44.71	(−6%)	20809	0	0
toString	1.62	(−7%)	0	0	0
isCharType	1.88	(+10%)	0	0	0
visitTryCatchStmt	3.86	(+30%)	3	0	0
binaryNumericPromotion	2.75	(+16%)	10	0	0
Parse	13.45	(+81%)	68	0	0
getRootInterface	10.43	(+25%)	20	0	0
checkTypeDeclOfSig	105.01	(+64%)	353	0	0
checkTypeDeclElem	148.47	(+68%)	349	0	0
main	6.16	(+3%)	18	0	0
getNextPragma	194.10	(+42%)	15361	234	1
scanNumber	25.32	(+16%)	2700	22	1

FIG. 9.   Performance data for the small test suite with the pattern-element optimization disabled.

triangle near the middle of the figure is an anomaly that we do not understand: somehow turning off the mod-time optimization caused a proof to fail. In spite of this embarrassing mystery, the evidence in favor of the mod-time optimization is compelling, both for improving average performance and for steering clear of the worst case.
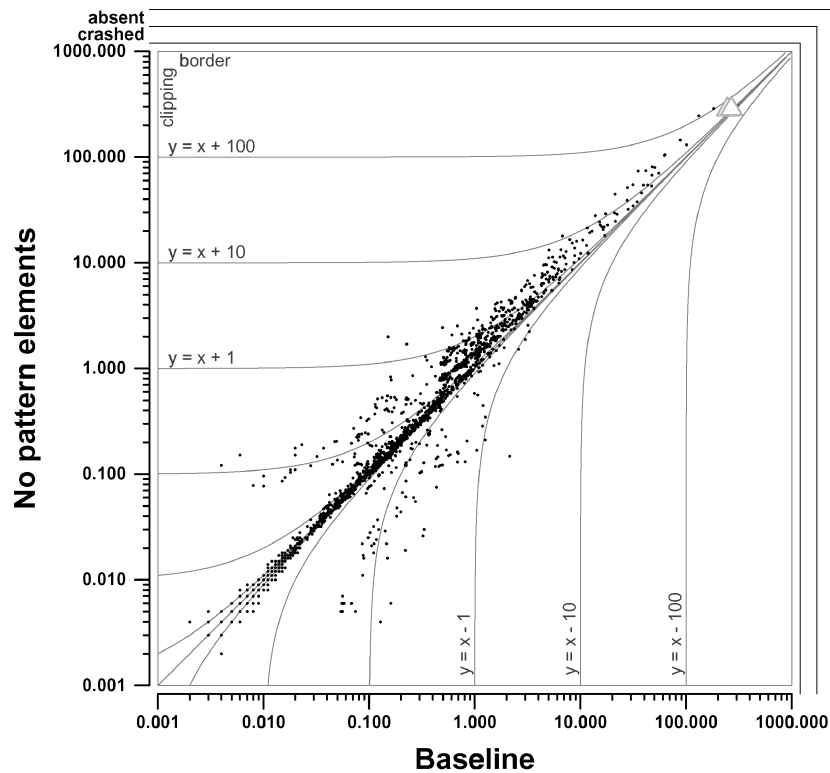
9.4. THE PATTERN-ELEMENT OPTIMIZATION.   As described in Section 5.4.3, Simplify uses the pattern-element optimization to avoid fruitlessly trying to match rules if no recent change to the E-graph can possibly have created new instances.

As shown in Figures 9 and 10, the results on the examples from program checking are similar to those for the mod-time optimization—either improvements (sometimes quite significant) or insignificant slowdowns. The domino examples suggest that the cost is significantly smaller than that for updating mod-times.

The mod-time and pattern-element optimizations have a common purpose—saving matching effort that cannot possibly yield new matches. Figure 11 shows what happens when both are disabled. About three quarters of the savings from the two optimizations overlap, but the remaining savings is significant enough to justify doing both.

9.5. SUBSUMPTION.   As described in Section 3.2, Simplify uses the subsumption heuristic, which asserts the negation of a literal upon backtracking from the case in which the literal was asserted. By default, Simplify uses subsumption by setting the status (§3.2) of the literal to **false**, and also, if it is a literal of the E-graph (except for nonbinary distinctions (§2)) or of an ordinary theory, denying it by a call to the appropriate assert method. By setting an environment variable, we can cause Simplify to set the status of the literal, but not to assert its negation. Figure 12 shows that the performance data for this option are inconclusive. The data on the front end test suite (not shown) are also inconclusive.

9.6. MERIT PROMOTION.   As described in Section 5.2.1, Simplify uses merit promotion: When a *Pop* reduces the matching depth, Simplify promotes the highest scoring (§3.6) clause from the deeper matching level to the shallower level.

Checked 2331 routines in 5299 sec. (vs. 3841 sec. for baseline)
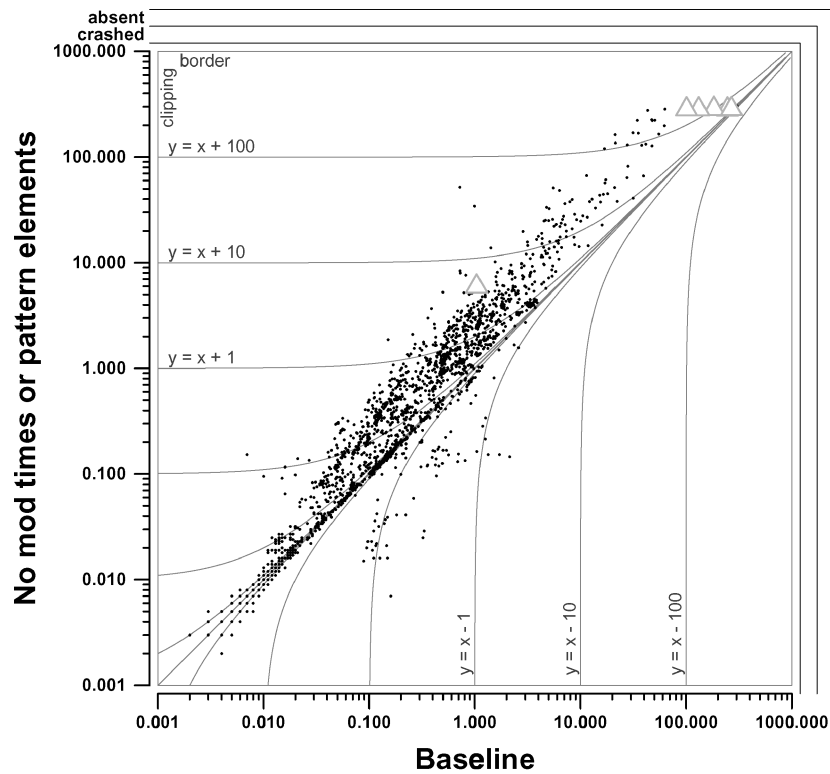Includes 3 timeouts.

FIG. 10.   Effect of disabling the pattern-element optimization on the front-end test suite.

Figures 13 and 14 show the effect of turning off merit promotion. In our small test suite, one method (cat) times out, and two methods (getNextPragma and scanNumber) show noticeable improvements. On the front-end suite, turning off promotion causes two methods to time out and improves the performance in a way that counts on only one (which in fact is getNextPragma). The total time increases by about ten percent.

We remark without presenting details that promote set size limits of 2, 10 (the default), and a million (effectively infinite for our examples) are practically indistinguishable on our test suites.

9.7. IMMEDIATE PROMOTION.   As described in Section 5.2.1, Simplify allows a matching rule to be designated for immediate promotion, in which case any instance of the rule is added to the current clause set instead of the pending clause set. In our test suites, the only immediately promoted rule is the nonunit select-of-store axiom (§2).

Figure 15 shows the effect of turning off immediate promotion on the small test suite. Immediate promotion provides a significant improvement on four test cases (cat, fastclose, simplex, and scanNumber), but a significant degradation on another (getNextPragma). The front-end test suite strengthens the case for the heuristic: Looking at the upper right corner of Figure 16 which is where the expensive verification conditions are, outliers above the diagonal clearly outnumber

Checked 2331 routines in 9544 sec. (vs. 3841 sec. for baseline)
Includes 6 timeouts.

FIG. 11.   Effect of disabling both the mod-time and pattern-element optimizations on the front-end test suite.

benchmark	time	(change)	splits	incs	max
addhi	3.57	(+5%)	116	4	1
cat	6.43	(−37%)	134	5	2
fastclose	2.56	(+8%)	70	0	0
frd-seek	5.62	(−13%)	146	0	0
simplex	28.70	(+2%)	142	0	0
domino6x4x	1.96	(−12%)	1194	0	0
domino6x6x	41.45	(−13%)	20809	0	0
toString	1.75	(+0%)	0	0	0
isCharType	1.84	(+8%)	0	0	0
visitTryCatchStmt	2.89	(−3%)	3	0	0
binaryNumericPromotion	2.24	(−6%)	10	0	0
Parse	7.41	(−0%)	75	0	0
getRootInterface	8.31	(−0%)	20	0	0
checkTypeDeclOfSig	63.18	(−1%)	353	0	0
checkTypeDeclElem	86.86	(−2%)	349	0	0
main	6.14	(+3%)	18	0	0
getNextPragma	122.92	(−10%)	15345	234	1
scanNumber	20.71	(−5%)	2700	22	1

FIG. 12.   Performance data for the small test suite with subsumption disabled.

benchmark	time	(change)	splits	incs	max
addhi	3.43	(+1%)	84	6	1
cat	>500.00	(>+4000%)	10900	4770	2
fastclose	2.59	(+9%)	70	0	0
frd-seek	6.56	(+2%)	171	1	1
simplex	28.26	(+0%)	134	0	0
domino6x4x	2.09	(−6%)	1194	0	0
domino6x6x	47.75	(+0%)	20809	0	0
toString	1.81	(+3%)	0	0	0
isCharType	1.68	(−2%)	0	0	0
visitTryCatchStmt	3.05	(+2%)	3	0	0
binaryNumericPromotion	2.33	(−2%)	10	0	0
Parse	7.44	(+0%)	70	0	0
getRootInterface	8.32	(−0%)	20	0	0
checkTypeDeclOfSig	63.43	(−1%)	353	0	0
checkTypeDeclElem	87.96	(−0%)	349	0	0
main	6.06	(+1%)	18	0	0
getNextPragma	58.72	(−57%)	3085	98	1
scanNumber	18.10	(−17%)	2000	46	1

FIG. 13.   Performance data for the small test suite with merit promotion disabled.
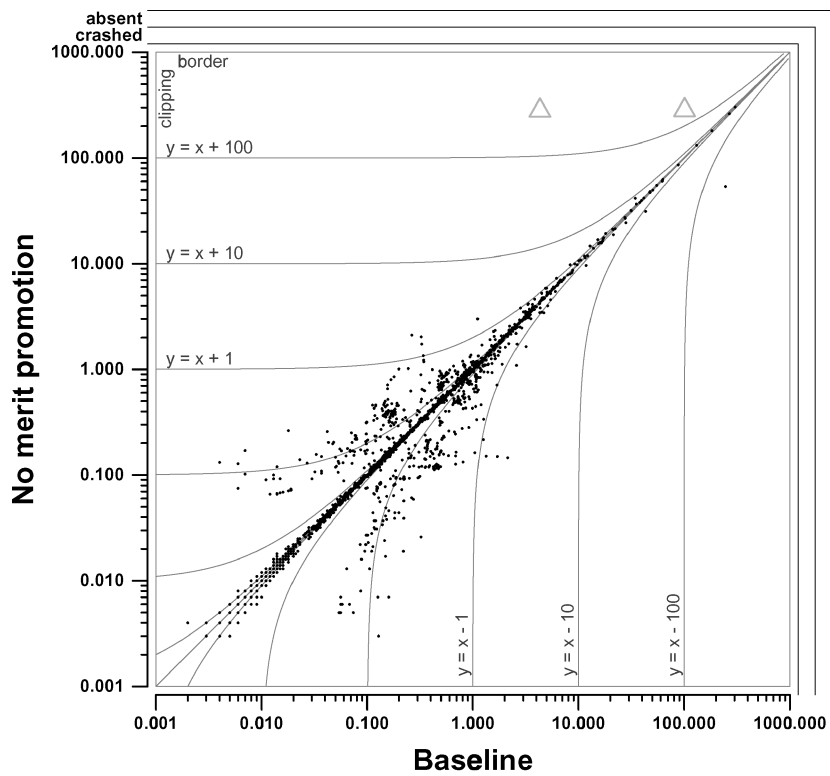
Checked 2331 routines in 4121 sec. (vs. 3841 sec. for baseline)
Includes 3 timeouts.
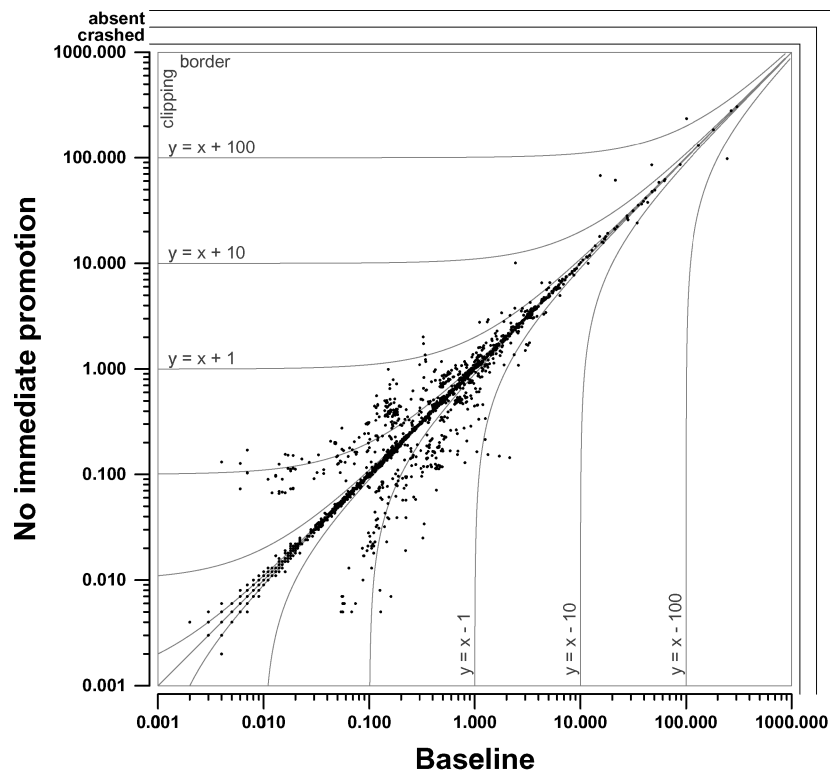
FIG. 14.   Effect of disabling merit promotion on the front-end test suite.

benchmark	time	(change)	splits	incs	max
addhi	3.80	(+12%)	97	6	1
cat	16.42	(+60%)	515	10	2
fastclose	4.45	(+88%)	225	8	1
frd-seek	6.58	(+2%)	171	1	1
simplex	76.99	(+173%)	344	29	1
domino6x4x	2.25	(+1%)	1194	0	0
domino6x6x	47.59	(−0%)	20809	0	0
toString	1.79	(+2%)	0	0	0
isCharType	1.84	(+8%)	0	0	0
visitTryCatchStmt	3.06	(+3%)	3	0	0
binaryNumericPromotion	2.28	(−4%)	10	0	0
Parse	6.68	(−10%)	65	28	2
getRootInterface	8.32	(−0%)	20	0	0
checkTypeDeclOfSig	63.64	(−1%)	353	0	0
checkTypeDeclElem	87.85	(−1%)	349	0	0
main	5.99	(+0%)	18	0	0
getNextPragma	96.24	(−30%)	9572	218	1
scanNumber	61.95	(+185%)	7286	90	1

FIG. 15.   Performance data for the small test suite with immediate promotion disabled.

Checked 2331 routines in 3974 sec. (vs. 3841 sec. for baseline)
Includes 1 timeout.

FIG. 16.   Effect of disabling immediate promotion on the front-end test suite.

benchmark	time	(change)	splits	incs	max
addhi	228.38	(+6637%)	560	5	1
cat	>500.00	(>+4000%)	567	4	1
fastclose	crash				
frd-seek	crash				
simplex	>500.00	(>+1000%)	0	0	0
domino6x4x	2.12	(−5%)	1194	0	0
domino6x6x	47.29	(−1%)	20809	0	0
toString	2.05	(+17%)	0	0	0
isCharType	2.08	(+22%)	0	0	0
visitTryCatchStmt	3.27	(+10%)	3	0	0
binaryNumericPromotion	3.45	(+45%)	10	0	0
Parse	>500.00	(>+6000%)	1647	49	1
getRootInterface	10.21	(+22%)	18	0	0
checkTypeDeclOfSig	50.01	(−22%)	286	0	0
checkTypeDeclElem	111.86	(+27%)	340	0	0
main	7.66	(+28%)	19	0	0
getNextPragma	>500.00	(> +200%)	418	4	1
scanNumber	>500.00	(>+2000%)	19205	44	1

FIG. 17.   Performance data for the small test suite with the activation heuristic disabled (so that triggers are matched against all E-nodes, rather than active E-nodes only).

outliers below the diagonal. Furthermore, the heuristic reduces the total time on the front end test suit by about five percent.

We remark without presenting details that consecutive immediately promoted split limits of 1, 2, 10 (the default), and a million (effectively infinite for our examples) are practically indistinguishable on our test suites.

9.8. ACTIVATION.   As described in Section 5.2, Simplify uses the activation heuristic, which sets the *active* bit in those E-nodes that represent subterms of atomic formulas (§2) that are currently asserted or denied, and restricts the matcher (§5.1) to find only those instances of matching rule triggers that lie in the active portion of the E-graph.

Figure 17 shows that the effect of turning off activation on the small test suite is very bad. Five of the tests time out, two crash with probable matching loops (see Section 5.1), several other tests slow down significantly, and only one (`checkType-DeclOfSig`) shows a significant improvement. In the front-end test suite (not shown) thirty-two methods time out, eighteen crash with probable matching loops, and the cloud of dots vote decisively for the optimization.

9.9. SCORING.   As described in Section 3.6, Simplify uses a scoring heuristic, which favors case splits that have recently produced contradictions. Figures 18 and 19 show that scoring is effective.

Of the problems in the small test suite derived from program checking, scoring helps significantly on six and hurts on none. The results are mixed for the two domino problems.
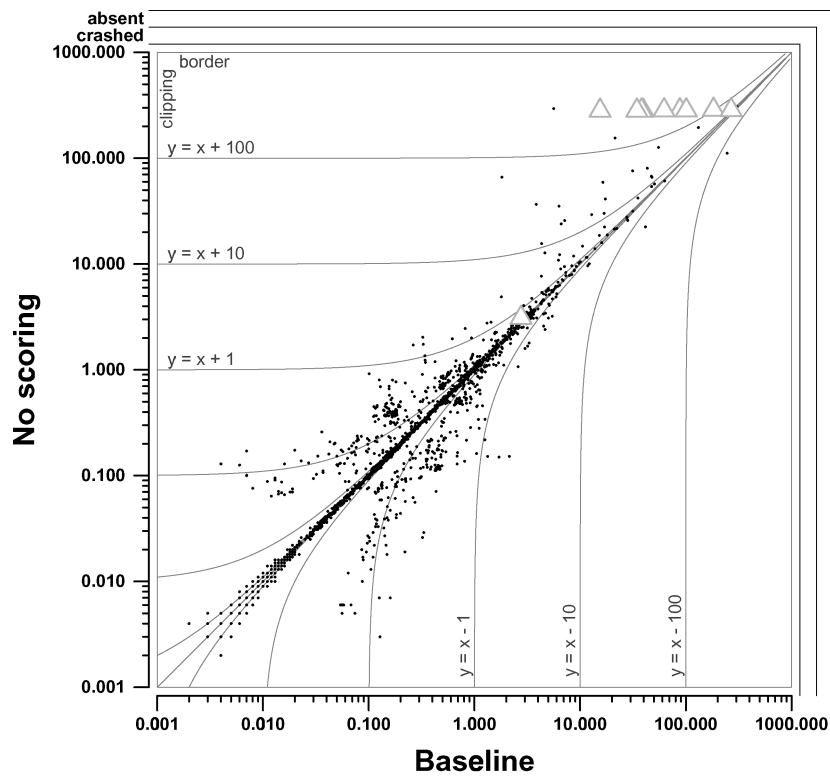
On the front-end test suite, scoring had mixed effects on cheap problems, but was very helpful for expensive problems. With scoring off ten problems time out, and the total time increases by sixty percent.

9.10. E-GRAPH TESTS.   As described in Section 4.6, Simplify uses E-graph tests, which checks the E-graph data structure, instead of the literal status fields only, when refining clauses. Figures 20 and 21 show that the E-graph test is effective, especially on expensive problems.

benchmark	time	(change)	splits	incs	max
addhi	9.53	(+181%)	1216	6	1
cat	36.84	(+259%)	1189	14	2
fastclose	2.43	(+3%)	70	0	0
frd-seek	12.47	(+93%)	424	1	1
simplex	28.74	(+2%)	143	0	0
domino6x4x	0.91	(−59%)	393	0	0
domino6x6x	56.62	(+19%)	25571	0	0
toString	1.76	(+1%)	0	0	0
isCharType	1.71	(+0%)	0	0	0
visitTryCatchStmt	3.11	(+4%)	3	0	0
binaryNumericPromotion	2.31	(−3%)	10	0	0
Parse	7.41	(−0%)	70	0	0
getRootInterface	8.41	(+1%)	20	0	0
checkTypeDeclOfSig	>500.00	(>+600%)	6782	0	0
checkTypeDeclElem	348.15	(+294%)	977	0	0
main	5.99	(+0%)	17	0	0
getNextPragma	117.85	(−14%)	6040	96	1
scanNumber	153.28	(+604%)	8676	16	1

FIG. 18.   Performance data for the small test suite with scoring disabled.
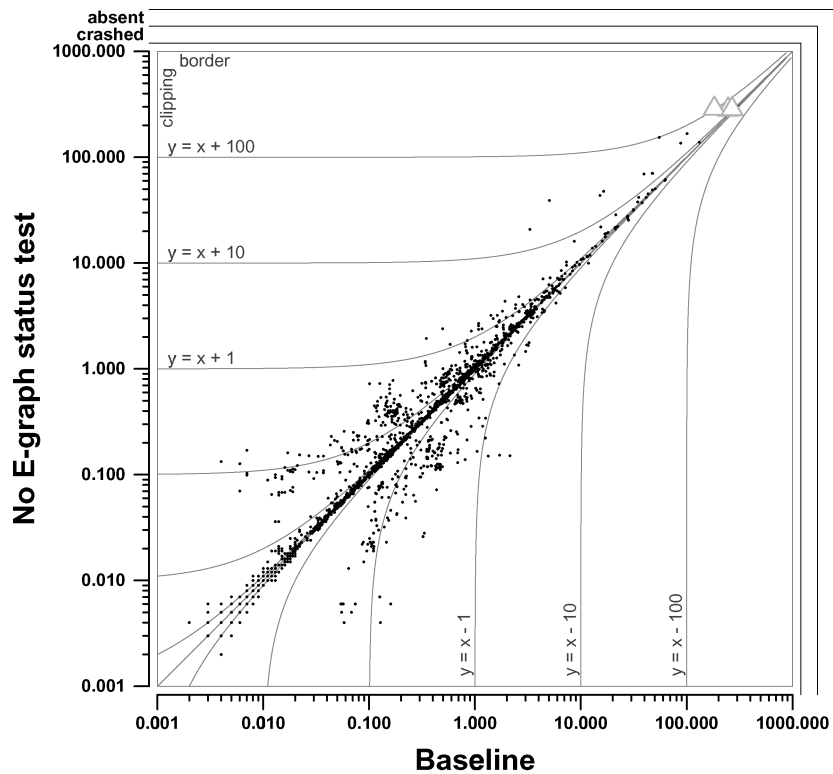

Checked 2331 routines in 6597 sec. (vs. 3841 sec. for baseline)
Includes 10 timeouts.

FIG. 19.   Effect of disabling scoring on the front-end test suite.

benchmark	time	(change)	splits	incs	max
addhi	3.78	(+12%)	143	8	1
cat	34.62	(+238%)	1282	18	3
fastclose	2.67	(+13%)	77	0	0
frd-seek	8.36	(+30%)	318	5	1
simplex	32.69	(+16%)	147	0	0
domino6x4x	2.24	(+0%)	1194	0	0
domino6x6x	46.56	(−2%)	20809	0	0
toString	1.79	(+2%)	0	0	0
isCharType	1.70	(−1%)	0	0	0
visitTryCatchStmt	2.96	(−1%)	3	0	0
binaryNumericPromotion	2.37	(−0%)	10	0	0
Parse	8.19	(+10%)	83	0	0
getRootInterface	8.20	(−2%)	32	2	1
checkTypeDeclOfSig	62.37	(−3%)	357	0	0
checkTypeDeclElem	133.20	(+51%)	637	1	1
main	6.11	(+2%)	19	0	0
getNextPragma	>500.00	(>+200%)	65650	496	1
scanNumber	32.88	(+51%)	5291	40	1

FIG. 20.   Performance data for the small test suite with the E-graph status test disabled.

Checked 2331 routines in 4560 sec. (vs. 3841 sec. for baseline)
Includes 4 timeouts.

FIG. 21.   Effect of disabling the E-graph status test on the front-end test suite.

benchmark	time	(change)	splits	incs	max
addhi	3.62	(+7%)	82	4	1
cat	10.82	(+6%)	253	6	2
fastclose	3.32	(+40%)	70	0	0
frd-seek	7.04	(+9%)	171	1	1
simplex	29.61	(+5%)	134	0	0
domino6x4x	2.15	(−4%)	1194	0	0
domino6x6x	47.12	(−1%)	20809	0	0
toString	3.29	(+88%)	0	0	0
isCharType	3.25	(+90%)	0	0	0
visitTryCatchStmt	4.35	(+46%)	3	0	0
binaryNumericPromotion	3.91	(+64%)	10	0	0
Parse	8.59	(+16%)	70	0	0
getRootInterface	9.47	(+14%)	20	0	0
checkTypeDeclOfSig	66.11	(+3%)	353	0	0
checkTypeDeclElem	92.69	(+5%)	349	0	0
main	7.68	(+28%)	18	0	0
getNextPragma	143.21	(+5%)	15365	234	1
scanNumber	24.37	(+12%)	2700	22	1

FIG. 22. Performance data for the small test suite with distinction classes disabled (so that an n-ary distinction is translated into $\binom{n}{2}$ binary distinctions).

9.11. DISTINCTION CLASSES. As described in Section 7.1, Simplify uses distinction classes, a space-efficient technique for asserting the pairwise distinctness of a set of more than two items. Interestingly enough, Figures 22 and 23 show the performance effects of distinction classes are negligible. We do not know what causes the strange asymmetrical pattern of dots in the front end test suite.

9.12. LABELS. As described in Section 6.2, Simplify uses labels to aid in error localization. We have done two experiments to measure the cost of labels, one to compare labels versus no error localization at all, and one to compare labels versus error variables.

The first experiment demonstrates that Simplify proves valid labeled verification conditions essentially as fast as it proves valid unlabeled verification conditions. This experiment was performed on the small test suite by setting a switch that causes Simplify to discard labels at an early phase. Figure 24 shows the results: there were no significant time differences on any of the examples.

The second experiment compares labels with the alternative "error variable" technique described in Section 6.2. It was performed on the small test suite by manually editing the examples to replace labels with error variables. Figure 25 shows the results: in all but one of the examples where the difference was significant, labels were faster than error variables.

9.13. SIMPLEX REDUNDANCY FILTERING. As described in Section 8.3.1, Simplify uses Simplex redundancy filtering to avoid processing certain trivially redundant arithmetic assertions. Figure 26 shows the effect of disabling this optimization on the front-end test suite. The aggregate effect of the optimization is an improvement of about 10 percent.

The remaining subsections of this section do not evaluate particular options, but present measurements that we have made that seem to be worth recording.

9.14. FINGERPRINTING MATCHES. As described in Section 5.2, Simplify maintains a table of fingerprints of matches, which is used to filter out redundant matches, that is, matches that have already been instantiated on the current path (§3.1). The
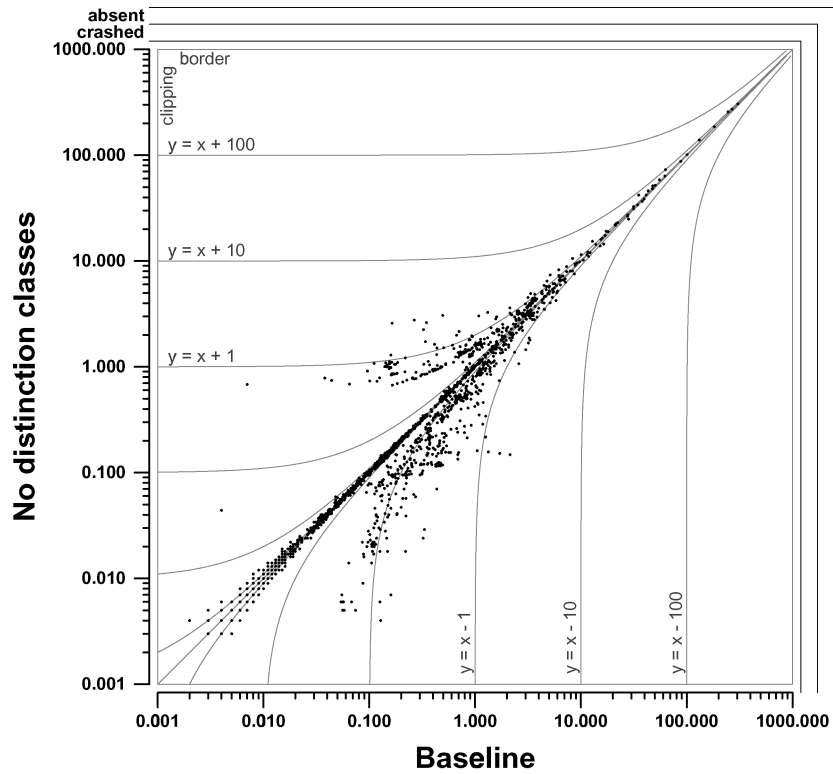
Checked 2331 routines in 3964 sec. (vs. 3841 sec. for baseline)
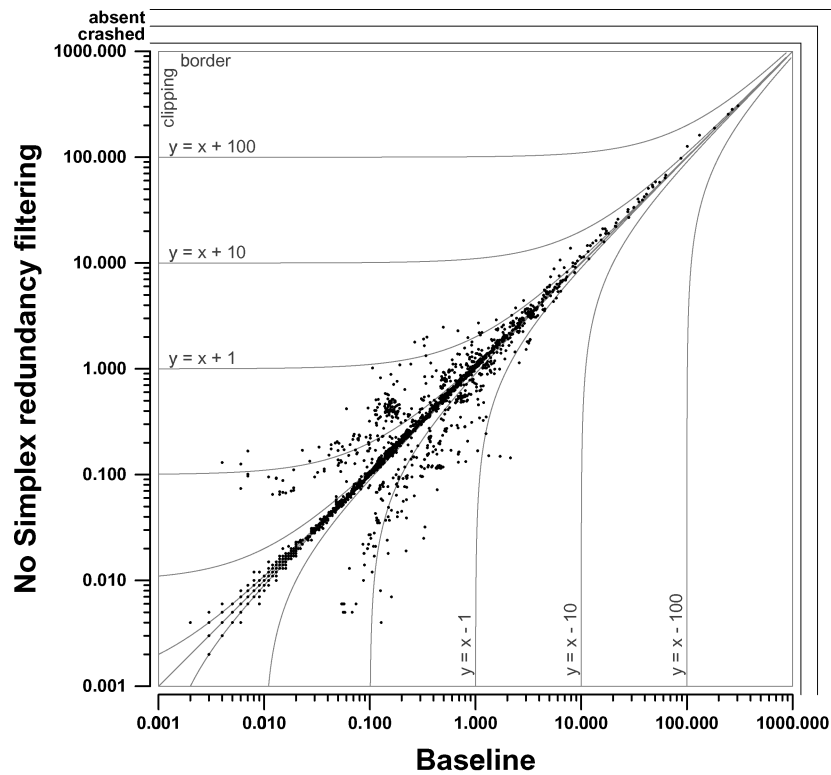Includes 1 timeout.

FIG. 23.   Effect of disabling distinction classes on the front-end test suite.

benchmark	time	(change)	splits	incs	max
addhi	3.41	(+1%)	82	4	1
cat	10.27	(+0%)	253	6	2
fastclose	2.62	(+11%)	68	0	0
frd-seek	6.43	(−0%)	170	1	1
simplex	28.37	(+0%)	134	0	0
domino6x4x	2.28	(+2%)	1194	0	0
domino6x6x	47.66	(−0%)	20809	0	0
toString	1.77	(+1%)	0	0	0
isCharType	1.75	(+2%)	0	0	0
visitTryCatchStmt	3.07	(+3%)	3	0	0
binaryNumericPromotion	2.36	(−1%)	10	0	0
Parse	7.38	(−1%)	70	0	0
getRootInterface	8.55	(+3%)	20	0	0
checkTypeDeclOfSig	62.45	(−3%)	353	0	0
checkTypeDeclElem	88.90	(+1%)	349	0	0
main	6.18	(+3%)	17	0	0
getNextPragma	136.83	(−0%)	15365	234	1
scanNumber	21.95	(+1%)	2700	22	1

FIG. 24.   Performance data for the small test suite with labels ignored.

benchmark	time	(change)	splits	incs	max
addhi	3.40	(+0%)	99	4	1
cat	10.29	(+0%)	273	6	2
fastclose	2.53	(+7%)	98	0	0
frd-seek	4.60	(−29%)	142	1	1
simplex	38.16	(+35%)	670	2	1
domino6x4x	2.22	(−0%)	1194	0	0
domino6x6x	47.39	(−1%)	20809	0	0
toString	1.60	(−9%)	0	0	0
isCharType	1.70	(−1%)	0	0	0
visitTryCatchStmt	3.37	(+13%)	8	0	0
binaryNumericPromotion	2.37	(−0%)	10	0	0
Parse	8.28	(+12%)	91	0	0
getRootInterface	9.88	(+18%)	53	0	0
checkTypeDeclOfSig	66.34	(+4%)	408	0	0
checkTypeDeclElem	246.26	(+179%)	962	24	1
main	5.97	(−0%)	42	0	0
getNextPragma	232.94	(+70%)	29424	236	1
scanNumber	26.50	(+22%)	4331	25	1

FIG. 25.   Performance data for the small test suite with labels replaced by error variables.

Checked 2331 routines in 4120 sec. (vs. 3841 sec. for baseline)
Includes 1 timeout.

FIG. 26.   Effect of disabling simplex redundancy filtering on the front-end test suite.

pivots:	0	1	2–3	4–7	8–15	16–31	32–63	64–127	128–255	256–511	512–1023
max(m, n)											
2–3	–	414	–	–	–	–	–	–	–	–	–
4–7	21302	1342	1	–	2	–	–	–	–	–	–
8–15	1123	12788	72	26	1110	34	–	–	–	–	–
16–31	14703	32969	2097	848	181	1128	1524	–	–	–	–
32–63	30073	46503	3550	390	146	1524	2045	2192	13	–	–
64–127	24282	39052	2650	213	63	246	1651	1395	1377	21	–
128–255	4107	9917	673	29	2	27	229	128	237	115	1
256–511	38	12	3	–	–	–	–	4	1	3	–
	95628	142997	9046	1506	1504	2959	5449	3719	1628	139	1
	(36%)	(54%)	(3.4%)	(.57%)	(.57%)	(1.1%)	(2.1%)	(1.4%)	(.62%)	(.05%)	(< .01%)

FIG. 27.   Distribution of the 264630 calls to *SgnOfMax* in the baseline run of the front-end test suite, according to tableau size (defined as max(m, n)) and number of pivots required.

mod-time (§5.4.2) and pattern-element (§5.4.3) optimizations reduce the number of redundant matches that are found, but even with these heuristics many redundant matches remain. On the program-checking problems in the small test suite, the fraction of matches filtered out as redundant by the fingerprint test ranges from 39% to 98% with an unweighted average of 71%. This suggests that there may be an opportunity for further matching optimizations that would avoid finding these redundant matches in the first place.

9.15. PERFORMANCE OF THE SIMPLEX MODULE.   To represent the Simplex tableau (§8.2), Simplify uses an ordinary m by n sequentially allocated array of integer pairs, even though in practice the tableau is rather sparse. It starts out sparse because the inequalities that occur in program checking rarely involve more than two or three terms. Although Simplify makes no attempt to select pivot elements so as to preserve sparsity, our dynamic measurements indicate that in practice and on the average, roughly 95% of the tableau entries are zero when *Pivot* is called. So a change to a sparse data structure (e.g., the one described by Knuth [1968, Sec. 2.2.6]) might improve performance, but more careful measurements would be required to tell for sure.

One important aspect of the performance of the Simplex module is the question of whether the number of pivots required to compute *SgnOfMax* in practice is as bad as its exponential worst case. Folklore says that it is not. Figure 27 presents evidence that the folklore is correct. We instrumented Simplify so that with each call to *SgnOfMax* it would log the dimensions of the tableau and the number of pivots performed. The figure summarizes this data. Ninety percent of the calls resulted in fewer than two pivots. Out of more than a quarter of a million calls to *SgnOfMax* performed on the front-end test suite, only one required more than 511 pivots (in fact 686).

9.16. PERFORMANCE OF THE E-GRAPH MODULE.   In this section, we present a few measurements we have made of the E-graph module. For each measurement, we report its minimum, maximum, and unweighted average value over the benchmarks of the small test suite. We included the domino benchmarks, although in some cases they were outliers.

The crucial algorithm for merging two equivalence classes and detecting new congruences is sketched in Section 4.2 and described in detail in Section 7. Simplify's *Sat* algorithm typically calls the *Merge* algorithm thousands of times per second. Over the small test suite, the rate ranges from 70 to 46000 merges/second and the unweighted average rate is 6100 merges/second.

Roughly speaking, the E-graph typically contains a few thousand active (§5.2) (concrete) (§7.1)) E-nodes. Over the small test suite, the maximum number of active E-nodes over the course of the proof ranged from 675 to 13000, and averaged 3400.

The key idea of the Downey–Sethi–Tarjan congruence-closure algorithm [Downey et al. 1980] is to find new congruences by rehashing the shorter of two parent lists, as described in Section 7.1. This achieves an asymptotic worst case time of $O(n \log n)$. The idea is very effective in practice. Over the examples of the small test suite, the number of E-nodes rehashed per *Merge* (ignoring rehashes in *UndoMerge*) ranged from 0.006 to 1.4, and averaged 0.75.

We chose to use the "quick find" union-find algorithm [Yao 1976] rather than one of the almost linear algorithms [Tarjan 1975] because the worst-case cost of this algorithm ($O(n \log n)$ on any path) seemed acceptable and the algorithm is easier to undo than union-find algorithms that use path compression. Over the examples of the small test suite, the average number of E-nodes rerooted per merge (ignoring rerooting during *UndoMerge*) ranged from 1.0 to 1.2, and averaged 1.05.

Over the examples of the small test suite, the fraction of calls to *Cons* that find and return an existing node ranged from 0.4 to 0.9, and averaged 0.79.

9.17. EQUALITY PROPAGATIONS. For the benchmarks in the small test suite, we counted the rate of propagated equalities per second between the E-graph and the Simplex module, adding both directions. The rate varies greatly from benchmark to benchmark, but is always far smaller than the rate of merges performed in the E-graph altogether. For the six benchmarks that performed no arithmetic, the rate is of course zero. For the others, the rate varies from a high of about a thousand propagated equalities per second (for cat, addhi, and frd-seek) to a low of two (for getRootInterface).

10. *Related and Future Work*

Like Simplify, the constraint solver Cassowary [Badros et al. 2001] implements a version of the Simplex algorithm with unknowns that are not restricted to be non-negative. The Cassowary technique, which the Cassowary authors attribute to Marriott and Stuckey [1998], is to use two tableaus: one tableau containing only the unknowns restricted to be non-negative, and a second tableau containing unrestricted unknowns. In their application (which doesn't propagate equalities, but does extract specific solutions), the use of two tableaus gives a performance advantage, since they needn't pivot the second tableau during satisfiability testing, but only when it is time to compute actual solution values for the unrestricted unknowns. Although Cassowary does not propagate equalities, Stuckey [1991] has described methods for detecting implicit equalities in a set of linear inequalities. Even in an application requiring equality propagation, the two-tableau approach might be advantageous, since the second tableau could be pivoted only when the dimension of the solution set diminished, making it necessary to search for new manifest equalities. Cassowary also supports constraint hierarchies (some constraints are preferences rather than requirements) and out-of-order (non-LIFO) removal of constraints.

A recent paper by Gulwani and Necula [2003] describes a an efficient randomized technique for incrementally growing a set of linear rational equalities and reporting any equalities of unknowns. Their technique, which reports two variables as equivalent if they are equal at each of a small set of pseudo-random points in the

solution flat, has a small chance of being unsound. However, it could safely be used as a filter tha might speed the search for new manifest equalities after a dimension reduction in a Simplex tableau.

The idea of theorem proving with cooperating decision procedures was introduced with the theorem prover of the Stanford Pascal Verifier. This prover, also named Simplify, was implemented and described by Greg Nelson and Derek Oppen [Nelson and Oppen 1979, 1980].

In addition to Simplify and its ancestor of the same name, several other automatic theorem provers based on cooperating decision procedures have been applied in program checking and related applications. These include the PVS prover from SRI [Owre et al. 1992], the SVC prover from David Dill's group at Stanford [Barrett et al. 1996], and the prover contained in George Necula's Touchstone system for proof-carrying code [Necula 1998; Necula and Lee 2000].

PVS includes cooperating decision procedures combined by Shostak's method [Shostak 1984]. It also includes a matcher to handle quantifiers, but the matcher seems not to have been described in the literature, and is said not to match in the E-graph [N. Shanker, 2003, Personal Communication].

SVC also uses a version of Shostak's method. It does not support quantifiers. It compensates for its lack of quantifiers by the inclusion of several new decision procedures for useful theories, including an extensional theory of arrays [Stump et al. 2001] and a theory of bit-vector arithmetic [Barrett et al. 1998].

Touchstone's prover adds proof-generation capabilities to a Simplify-like core. It doesn't handle general quantifiers, but does handle first-order Hereditary Harrop formulas.

During our use of Simplify, we observed two notable opportunities for performance improvements. First, the backtracking search algorithm that Simplify uses for propositional inference had been far surpassed by recently developed fast SAT solvers [Silva and Sakallah 1999; Zhang 1997; Moskewicz et al. 2001]. Second, when Simplify is in the midst of deeply nested case splits and detects an inconsistency with an underlying theory, the scoring (§3.6) heuristic exploits the fact that the most recent case split must have contributed to the inconsistency, but Simplify's decision procedures supply no information about which of the other assertions in the context contributed to the inconsistency.

In the past few years several systems hae been developed that exploit these opportunities by coupling modern SAT solving algorithms with domain-specific decision modules that are capable of reporting reasons for any inconsistencies that they detect. When the domain-specific decision modules detect an inconsistency in a candidate SAT solution, the reasons are recorded as new propositional clauses (variously referred to as "explicated clauses" [Flanagan et al. 2003], "conflict clauses" [Barrett et al. 2002a] and "lemmas on demand" [de Moura and Ruess 2002] that the SAT solver then uses to prune its search for further candidate solutions. It is interesting to note that a very similar idea occurs in a 1977 paper by Stallman and Sussman [Stallman and Sussman 1977], in which newly discovered clauses are called "NOGOOD assertions".

11. *Our Experience*

We would like to include a couple of paragraphs describing candidly what we have found it like to use Simplify.

First of all, compared to proof checkers that must be steered case by case through a proof that the user must design herself, Simplify provides a welcome level of automation.

Simplify is usually able to quickly dispatch valid conjectures that have simple proofs. For invalid conjectures with simple counterexamples, Simplify is usually able to detect the invalidity, and the label mechanism described in Section 6.2 is usually sufficient to lead the user to the source of the problem. These two cases cover the majority of conjectures encountered in our experience with ESC, and we suspect that they also cover the bulk of the need for automatic theorem proving in design-checking tools.

For more ambitious conjectures, Simplify may time out, and the diagnosis and fix are likely to be more difficult. It might be that what is needed is a more aggressive trigger (§5.1) on some matching rule (to find some instance that is crucial to the proof). Alternatively, the problem might be that the triggers are already too aggressive, leading to useless instances or even to matching loops. Or it might be that the triggers are causing difficulty only because of an error in the conjecture. There is no easy way to tell which of these cases applies.

In the ESC application, we tuned the triggers for axioms in the background predicate (§2) so that most ESC users do not have to worry about triggers. But ambitious users of ESC who include many quantifiers in their annotations may well find that Simplify is not up to the task of choosing appropriate triggers.

12. *Conclusions*

Our main conclusion is that the Simplify approach is very promising (nearly practical) for many program-checking applications. Although Simplify is a research prototype, it has been used in earnest by groups other than our own. In addition to ESC, Simplify has been used in the Slam device driver checker developed at Microsoft [Ball et al. 2001] and the KeY program checker developed at Karlsruhe University [P. H. Schmitt, 2003, Personal Communication; Ahrendt et al. 2003]. One of us (Saxe) has used Simplify to formally verify that a certain feature of the Alpha multiprocessor memory model is irrelevant in the case of programs all of whose memory accesses are of uniform granularity. Another of us (Detlefs) has used Simplify to verify the correctness of a DCAS-based lock-free concurrent double-ended-queue. The data structure is described in the literature [Agesen et al. 2000] and the Simplify proof scripts are available on the web [Detlefs and Moir 2000]. The Denali 1 research project [Joshi et al. 2002] (a superoptimizer for the Alpha EV6 architecture) used Simplify in its initial feasibility experiments to verify that optimal machine code could be generated by automatic theorem-proving methods. The Denali 2 project (a superoptimizer for the McKinley implementation of IA-64) has used Simplify to check that axioms for a little theory of prefixes and suffixes are sufficient to prove important facts about the IA-64 extract and deposit instructions.

In particular, we conclude that the Nelson–Oppen combination method for reasoning about important convex theories works effectively in conjunction with the use of matching to reason about quantifiers. We have discussed for the first time some crucial details of the Simplify approach: undoing merge operations in the E-graph, correctly propagating equalities from the Simplex algorithm, the iterators for matching patterns in the E-graph, the optimizations that avoid fruitless matching

effort, the interactions between matching and the overall backtracking search, the notion of an ordinary theory and the interface to the module that reasons about an ordinary theory.

Appendix

A. *Index of Selected Identifiers*

We list here each identifier that is used in this paper outside the top-level section in which it is defined, together with a brief reminder of its meaning and a reference to its section of definition.

$e.active$: True if E-node e represents a term in a current assertion (Sec. 5.2)

$T.Asserted$: Conjunction of the currently asserted T-literals (Sec. 4.4)

$AssertEQ$: Assert equality between terms represented as E-nodes (Sec. 7.2)

$AssertLit$: Assert a literal (Sec. 3.1)

$AssertZ$: Assert that a formal affine sum of Simplex unknowns equals 0 (Sec. 8.4)

$children[u]$: Children of node u in the term DAG (Sec. 4.2)

cls: Clause set of the current context (Sec. 3.1)

$Cons$: Constructor for binary E-nodes (Sec. 7.4)

DEFPRED: Declare, and optionally define, a quasi-relation (Sec. 2)

DISTINCT: n-ary distinction operator (Sec. 2)

E-node: E-node data type (Sec. 4.2)

$u.enode$: E-node associated with unknown u of an ordinary theory (Sec. 4.4)

$e.id$: Unique numerical identifier of E-node e (Sec. 7.1)

$r.lbls$: Set of labels (function symbols) of active E-nodes in the class with root r (Sec. 5.4.4)

$lits$: Literal set of the current context (Sec. 3.1)

$Merge$: Merge equivalence classes in the E-graph (Sec. 7.2)

$Pivot$: Pivot the Simplex tableau (Sec. 8.3)

$r.plbls$: Set of labels of active (term DAG) parents of E-nodes in the class with root r (Sec. 5.4.4)

Pop: Restore a previously saved context (Sec. 3.1)

$T.Propagated$: Conjunction of the equalities currently propagated from theory T (Sec. 4.4)

$Push$: Save the current context (Sec. 3.1)

$Refine$: Perform width reduction, clause elimination, and unit assertion (Sec. 3.2); extended to perform matching (Sec. 5.2)

$refuted$: If true, the current context is inconsistent (Sec. 3.1)

$e.root$: Root of E-node e's equivalence class (Sec. 4.2)

Sat: Determine whether the current context is satisfiable (Sec. 3.2); respecified (Sec. 3.4.1)

$select$: Access operator for maps (arrays) (Sec. 2)

$SgnOfMax$: Determine sign of maximum of a Simplex unknown (Sec. 8.3)

Simplex:	Module for reasoning about arithmetic (Sec. 4.3)
store:	Update operator for maps (arrays) (Sec. 2)
r.T_unknown:	*T.Unknown* associated with root E-node *r* (Sec. 4.4)
@true:	Constant used to reflect **true** as an individual value (Sec. 2)
UndoMerge:	Undo effects of *Merge* (Sec. 7.3)
T.Unknown:	Unknown of the theory *T* (Sec. 4.4)
T.UnknownForEnode:	Find or create a *T.Unknown* associated with an E-node (Sec. 4.4)

REFERENCES

AGESEN, O., DETLEFS, D. L., FLOOD, C. H., GARTHWAITE, A. T., MARTIN, P. A., SHAVIT, N. N., AND STEEL, JR., G. L. 2000. Dcas-based concurrent deques. In *ACM Symposium on Parallel Algorithms and Architectures*. 137–146.

AHRENDT, W., BAAR, T., BECKERT, B., BUBEL, R., GIESE, M., HÄHNLE, R., MENZEL, W., MOSTOWSKI, W., ROTH, A., SCHLAGER, S., AND SCHMITT, P. H. 2003. The KeY tool. Technical report in computing science no. 2003–5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden. February.

BADROS, G. J., BORNING, A., AND STUCKEY, P. J. 2001. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction 8*, 4 (Dec.), 267–306.

BALL, T., MAJUMDAR, R., MILLSTEIN, T. D., AND RAJAMANI, S. K. 2001. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah. 203–213.

BARRETT, C. W. 2002. Checking validity of quantifier-free formulas in combinations of first-order theories. Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA. Available at `http://verify.stanford.edu/barrett/thesis.ps`.

BARRETT, C. W., DILL, D. L., AND LEVITT, J. 1996. Validity checking for combinations of theories with equality. In *Proceedings of Formal Methods In Computer-Aided Design*. 187–201.

BARRETT, C. W., DILL, D. L., AND LEVITT, J. R. 1998. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Design Automation Conference*. San Francisco, CA.

BARRETT, C. W., DILL, D. L., AND STUMP, A. 2002a. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, E. Brinksma and K. G. Larsen, Eds. Number 2404 in Lecture Notes in Computer Science. Springer-Verlag. Copenhagen.

BARRETT, C. W., DILL, D. L., AND STUMP, A. 2002b. A generalization of Shostak's method for combining decision procedures. In *Frontiers of Combining Systems (FROCOS)*. Lecture Notes in Artificial Intelligence. Springer-Verlag. Santa Margherita di Ligure, Italy.

BIBEL, W., AND EDER, E. 1993. Methods and calculi for deduction. In *Handbook of Logic in Artificial Intelligence and Logic Programming—Vol 1: Logical Foundations.*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Clarendon Press, Oxford, 67–182.

CHVATAL, V. 1983. *Linear Programming*. W H Freeman & Co.

CONCHON, S., AND KRSTIĆ, S. 2003. Strategies for combining decision procedures. In *Proceedings of the 9th International Conferences on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Lecture Notes in Computer Science, vol. 2619. Springer Verlag, 537–553.

CROCKER, S. 1988. Comparison of Shostak's and Oppen's solvers. Unpublished manuscript.

DANTZIG, G. B. 1963. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ.

DE MOURA, L., AND RUESS, H. 2002. Lemmas on demand for satisfiability solvers. In *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing*.

DE MOURA, L. M., AND RUESS, H. 2004. An experimental evaluation of ground decision procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, R. Alur and D. A. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer, 162–174. See `http://www.csl.sri.com/users/demoura/gdp-benchmarks.html` for benchmarks and additional results.

DETLEFS, D., AND MOIR, M. 2000. Mechanical proofs of correctness for dcas-based concurrent deques. Available at `http://research.sun.com/jtech/pubs/00-deque1-proof.html`.

DETLEFS, D., NELSON, G., AND SAXE, J. B. 2003a. Simplify benchmarks. Available at `http://www.hpl.hp.com/research/src/esc/simplify_benchmarks.tar.gz`. These benchmarks are also available in the appendix to the online version of this article, available via the ACM Digital Library.

DETLEFS, D., NELSON, G., AND SAXE, J. B. 2003b. Simplify source code. Available at `http://www.research.compaq.com/downloads.html` as part of the Java Programming Toolkit Source Release.

DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. 1998. Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, USA. December. Available at `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-159.html`.

DOWNEY, P. J., SETHI, R., AND TARJAN, R. E. 1980. Variations on the common subexpression problem. *JACM 27*, 4 (Oct.), 758–771.

FLANAGAN, C., JOSHI, R., OU, X., AND SAXE, J. B. 2003. Theorem proving using lazy proof explication. In *Proceedings of the 15th International Conference on Computer Aided Verification*. 355–367.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language Design and Implementation (PLDI'02)*. Berlin, 234–245.

GALLER, B. A., AND FISCHER, M. J. 1964. An improved equivalence algorithm. *CACM 7*, 5, 301–303.

GANZINGER, H. 2002. Shostak light. In *Proceedings 18th International Conference on Automated Deduction (CADE 18)*, A. Voronkov, Ed. Lecture Notes in Computer Science, vol. 2392. Springer, Copenhagen, 332–346.

GANZINGER, H., RUESS, H., AND SHANKAR, N. 2004. Modularity and refinement in inference systems. CSL Technical Report CSL-SRI-04-02, SRI. Dec. Available at ftp://ftp.csl.sri.com/pub/users/shankar/modularity.ps.gz.

GUERRA E SILVA, L., MARQUES-SILVA, J., AND SILVEIRA, L. M. 1999. Algorithms for solving boolean satisfiability in combinational circuits. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*. Munich, 526–530.

GULWANI, S., AND NECULA, G. C. 2003. A randomized satisfiability procedure for arithmetic and uninterpreted function symbols. In *19th International Conference on Automated Deduction*. LNCS, vol. 2741. Springer-Verlag, 167–181.

JOSHI, R., NELSON, G., AND RANDALL, K. 2002. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM 2002 Conference on Programming Language Design and Implementation*. Berlin, 304–314.

KNUTH, D. E. 1968. *The Art of Computer Programming—Vol. 1 Fundamental Algorithms*. Addison Wesley, Reading, MA. 2nd ed. 1973.

KNUTH, D. E., AND SCHÖNHAGE, A. 1978. The expected linearity of a simple equivalence algorithm. *Theoretical Computer Science 6*, 3 (June), 281–315.

KOZEN, D. 1977. Complexity of finitely presented algebras. In *Proceedings Ninth STOC*. 164–177.

KRSTIĆ, S., AND CONCHON, S. 2003. Canonization for disjoint unions of theories. In *Proceedings of the 19th International Conference on Automated Deduction (CADE-19)*, F. Baader, Ed. Lecture Notes in Computer Science, vol. 2741. Springer Verlag.

LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, J. E. B., SCHAFFERT, C., SCHEIfLER, R., AND SNYDER, A. 1981. *CLU Reference Manual*. Lecture Notes in Computer Science, vol. 114. Springer-Verlag, Berlin.

LOVELAND, D. W. 1978. *Automated Theorem Proving: A Logical Basis*. Elsevier Science.

MARCUS, L. 1981. A comparison of two simplifiers. Microver Note 94, SRI. January.

MARRIOTT, K., AND STUCKEY, P. J. 1998. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, MA.

MCCARTHY, J. 1963. Towards a mathematical science of computation. In *Information Processing: The 1962 IFIP Congress*. 21–28.

MILLSTEIN, T. 1999. Toward more informative ESC/Java warning messages. In Compaq SRC Technical Note 1999-003. Available at `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1999-003.html`.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*.

NECULA, G. C. 1998. Compiling with Proofs. Ph.D. thesis, Carnegie-Mellon University. Also available as CMU Computer Science Technical Report CMU-CS-98-154.

NECULA, G. C., AND LEE, P. 2000. Proof generation in the Touchstone theorem prover. In *Proceedings of the 17th International Conference on Automated Deduction*. 25–44.

NELSON, C. G. 1979. Techniques for program verification. Ph.D. thesis, Stanford University. A revised version of this thesis was published as a Xerox PARC Computer Science Laboratory Research Report [Nelson 1981].

NELSON, G. 1981. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC Computer Science Laboratory. June.

NELSON, G. 1983. Combining satisfiability procedures by equality-sharing. In *Automatic Theorem Proving: After 25 Years*, W. W. Bledsoe and D. W. Loveland, Eds. American Mathematical Society, 201–211.

NELSON, G., AND OPPEN, D. C. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1*, 2 (Oct.), 245–257.

NELSON, G., AND OPPEN, D. C. 1980. Fast decision procedures based on congruence closure. *JACM 27*, 2 (April), 356–364.

OWRE, S., RUSHBY, J. M., AND SHANKAR, N. 1992. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, D. Kapur, Ed. Lecture Notes in Artificial Intelligence, vol. 607. Springer-Verlag, Saratoga, NY, 748–752. Available at `http://www.csl.sri.com/papers/cade92-pvs/`.

RABIN, M. O. 1981. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University.

RUESS, H., AND SHANKAR, N. 2001. Deconstructing Shostak. In *Proceedings of the LICS 2001*. 10–28.

SCHMITT, P. H. 2003. Personal communication (email message to Greg Nelson).

SHANKAR, N. 2003. Personal communication (email message to James B. Saxe).

SHANKAR, N., AND RUESS, H. 2002. Combining Shostak theories. Invited paper for Floc'02/RTA'02. Available at `ftp://ftp.csl.sri.com/pub/users/shankar/rta02.ps`.

SHOSTAK, R. E. 1979. A practical decision procedure for arithmetic with function symbols. *JACM 26*, 2 (April), 351–360.

SHOSTAK, R. E. 1984. Deciding combinations of theories. *JACM 31*, 1, 1–12. See also [Barrett et al. 2002b; Ruess and Shankar 2001].

SILVA, J. M., AND SAKALLAH, K. A. 1999. GRASP: A search algorithm for propositionsal satisfiability. *IEEE Transactions on Computers 48*, 5 (May), 506–521.

STALLMAN, R. M., AND SUSSMAN, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence 9*, 2 (Oct.), 135–196.

STUCKEY, P. J. 1991. Incremental linear constraint solving and detection of implicit equalities. *ORSA Journal on Computing 3*, 4, 269–274.

STUMP, A., BARRETT, C., DILL, D., AND LEVITT, J. 2001. A decision procedure for an extensional theory of arrays. In *16th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 29–37.

TARJAN, R. E. 1975. Efficiency of a good but not linear set union algorithm. *JACM 22*, 2, 215–225.

TINELLI, C., AND HARANDI, M. T. 1996. A new correctness proof of the Nelson-Oppen combination procedure. In *Frontiers of Combining Systems: Proceedings of the 1st International Workshop*, F. Baader and K. U. Schulz, Eds. Kluwer Academic Publishers, Munich, 103–120.

YAO, A. 1976. On the average behavior of set merging algorithms. In *8th ACM Symposium on the Theory of Computation*. 192–195.

ZHANG, H. 1997. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*. 272–275.