Simplify: A Theorem Prover for Program Checking

David Detlefs[1], Greg Nelson, James B. Saxe
Systems Research Center
HP Laboratories Palo Alto

HPL-2003-148
July 16th , 2003*

E-mail: david.detlefs@sun.com, gnelson@hp.com, jim.saxe@hp.com

theorem-proving, decision procedures, program checking

This paper 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 paper describes two techniques, labels and counterexample contexts, for helping the user to determine the reason that a false conjecture is false. The paper includes detailed performance figures on conjectures derived from realistic program-checking problems

* Internal Accession Date Only

[1] Currently affiliated with Sun Microsystems

Simplify: A Theorem Prover for Program Checking

DAVID DETLEFS[1] and GREG NELSON and JAMES B. SAXE
Hewlett-Packard Systems Research Center

This paper 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 paper describes two techniques, labels and counterexample contexts, for helping the user to determine the reason that a false conjecture is false. The paper 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

Contents

1 Introduction **3**

2 Simplify's built-in theory **5**

3 The search strategy **8**
3.1 The interface to the context 8
3.2 The *Sat* procedure 9
3.3 Equisatisfiable CNF 13
3.4 Avoiding exponential mixing with lazy CNF 14
3.4.1 Details of lazy CNF 15
3.4.2 Comparison with resolution 19
3.5 Goal proxies and asymmetrical implication 20
3.6 Scoring clauses 21
3.7 Using the subsumption heuristic for proxy literals 22

4 Domain-specific decision procedures **23**

[1]David Deltefs is currently affiliated with Sun Microsystems

Author's addresses: David Detlefs, Mailstop UBUR02-311, Sun Microsystems Laboratories, One Network Drive, Burlington, MA 01803-0902. Email: david.detlefs@sun.com Greg Nelson, System Research Center, Mailstop 1250, Hewlett-Packard Labs, 1501 Page Mill Rd., Palo Alto, CA 94303. Email: gnelson@hp.com James B. Saxe, System Research Center, Mailstop 1250, Hewlett-Packard Labs, 1501 Page Mill Rd., Palo Alto, CA 94303. Email: jim.saxe@hp.com

4.1 Equality sharing 24
4.2 The E-graph module 26
4.3 The Simplex module 29
4.4 Ordinary theories and the special role of the E-graph 29
4.5 Connecting the E-graph with the ordinary theories 31
4.6 Width reduction with domain-specific literals 33
4.7 The theory of partial orders 34

5 Quantifiers 35

5.1 Searching and matching 38
5.1.1 The matching depth heuristic. 41
5.2 Quantifiers to matching rules 42
5.2.1 Simple story 42
5.2.2 Gory details. 44
5.3 How triggers are matched 46
5.3.1 Matching iterators 46
5.3.2 The mod-time matching optimization 50
5.3.3 The pattern-element matching optimization 53
5.3.4 Implementation of the pattern-element optimization 55
5.3.5 Mod-times and pattern elements and multi-triggers 56

6 Reporting errors 58

6.1 Error context reporting 59
6.2 Error localization 59
6.3 Multiple counterexamples 62

7 The E-graph in detail 62

7.1 Data structures and invariants 62
7.2 *AssertEq* and *Merge* 67
7.3 *UndoMerge* 71
7.4 *Cons* 79
7.5 Three final fine points 80

8 The Simplex module in detail 81

8.1 The interface 82
8.2 The tableau 83
8.3 Testing consistency 86
8.3.1 Simplex redundancy filtering 91
8.4 Propagating equalities 92
8.5 Undoing tableau operations 97
8.6 Integer heuristics 98

9 Performance measurements 99

9.1 Plunging 101
9.2 The mod-time optimization 103
9.3 The pattern-element optimization 104
9.4 Subsumption 105

9.5 Merit promotion 105
9.6 Immediate promotion 106
9.7 Activation 110
9.8 Scoring 110
9.9 E-graph tests 112
9.10 Distinction classes 112
9.11 Labels 112
9.12 Simplex redundancy filtering 115
9.13 Fingerprinting matches 115
9.14 Performance of the Simplex module 115
9.15 Performance of the E-graph module 116
9.16 Equality propagations 117

10 Related and future work **117**

11 Our experience **118**

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. The ESC methodology is first to process source code with a *verification condition generator*, which produces first-order formulas asserting the absence of the targeted errors, and then to submit those *verification conditions* to the theorem prover. Although designed for ESC, the prover is interesting in its own right and has been used for purposes other than ESC. Several examples are listed in the conclusions.

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 McCarthy's functions for updating and accessing arrays are also pre-defined. 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 paper 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. The prover 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:

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

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

$$\begin{array}{l} x < y \\ x - 1 \geq y \\ x \geq y + 2 \end{array}$$

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 $x - 1 \geq y \vee x \geq y + 2$
  first case, assume $x - 1 \geq y$
    discover the inconsistency of $x < y \ \wedge \ x - 1 \geq y$
    backtrack from the first case (discard the assumption $x - 1 \geq y$)
  second case, assume $x \geq y + 2$
    discover the inconsistency of $x < y \ \wedge \ x \geq y + 2$
    backtrack form 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. The prover 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 the prover 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.

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 paper.

The input to our theorem prover is a formula of untyped first-order logic with function symbols and equality. That is, the language includes the *propositional connectives* $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$; the *universal quantifier* $\forall$, and the *existential quantifier* $\exists$.

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:

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

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. In this paper, we will write $f[x]$ as shorthand for $\mathit{select}(f, x)$.

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.

Our theory is untyped, so that expressions that are intuitively mistyped, like $\mathit{select}(6, 2)$ or $\mathit{store}(a, i, x) + 3$ are legal, but the prover will not be able to prove anything non-trivial about such terms.

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 $\{\mathbf{true}, \mathbf{false}\}$, 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.

An *atomic formula* is a propositional variable, a propositional literal constant, or the application of a relation to a list of terms.

The strict distinction between formulas and terms is a feature of the classical treatment of first-order logic that we were treated to in our educations, 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

$$(\forall x : f(x) = @\mathbf{true} \;\Leftrightarrow\; g(x) = @\mathbf{true} \;\wedge\; h(x) = @\mathbf{true})$$

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

The DEFPRED facility has another, more general, form:

$$(\mathtt{DEFPRED}\ (r\ \langle args \rangle)\ body) \tag{1}$$

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

$$(\forall args : r(args) = @\mathbf{true} \;\Leftrightarrow\; body) \tag{2}$$

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. 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.

Simplify requires that its input be presented as a symbolic expression as in Lisp, but in this paper we will usually use more conventional mathematical notation.

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 $DISTINCT(t_1, \ldots, t_n)$ 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.

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

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

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

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 occurrence of a variable in the conjecture or query is said to *outermost* if it not in the scope of any quantifier. Outermost variables are implicitly universally quantified in the conjecture and implicitly existentially quantified in the query. A term is a *ground term* if all variables occurring in it are outermost variables.

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*).

3. THE 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 uninterpreted propositional variables.

3.1 The interface to the context

The prover uses a global resettable data structure called the *context* which represents the conjunction of the negated conjecture 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 *Sat* algorithm operates on the literal set through the following interface, called the *satisfiability interface*:

$$\begin{array}{rcl} \mathit{AssertLit}(P) & \equiv & \text{add the literal } P \text{ to } \mathit{lits} \text{ and possibly} \\ & & \quad \text{set } \mathit{refuted} \text{ if this makes } \mathit{lits} \text{ inconsistent} \\ \mathit{Push}() & \equiv & \text{save the state of the context} \\ \mathit{Pop}() & \equiv & \text{restore the most recently saved, but} \\ & & \quad \text{not-yet-restored, context} \end{array}$$

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 5.1) 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**, the prover would become unsound. Were it to be left **false** unnecessarily, the prover 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 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 interesting 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 below 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 negated conjecture $\neg G$, 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 consis-
tent, (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 that *Sat*, as given here, computes a *disjunctive normal form* for the context. If the prover 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. *Soundness*, the property that invalid conjecture are always refuted, is equivalent to the property that satisfiable queries are always found to be satisfiable, that is

(3′) if the context is satisfiable, at least one monome is output.

which is a weakened version of (3). Conditions (1) and (2) together guarantee that the prover, if it always terminates, is *complete*: if the conjecture is valid, then the prover will find no satisfying assignments for its negation, and thus will report the conjecture to be valid.

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 $\Rightarrow$ $l$] then
    delete $C$ from cls;
    return
  end;
  while $C$ contains a literal $l$ such that [lits $\Rightarrow$ $\neg l$] do
    delete $l$ from $C$
  end;
  if $C$ is empty then
```

```
        $refuted$ := **true**
    **elseif** $C$ 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 literals, it is easy to test whether $[lits \Rightarrow l]$: this condition is equivalent to $l \in lits$. Later in this paper, 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

$Refine(); Sat()$

meets the specification for *Sat*. Moreover, *Refine* has the heuristically desirable effects of

- narrowing clauses by removing literals that are inconsistent with *lits* and thus inconsistent with the context (called *width reduction*)
- removing clauses that are already satisfied by *lits* (called *clause elimination*), 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 unit assertion, then width reduction, then clause elimination. 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** $Sat() \equiv$
    enable refinement;
    $Refine()$;
    **if** $refuted$ **then return**
    **elsif** $cls$ 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 non-empty, 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 5.1 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 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 $Context.AssertLit(\neg\, l)$ 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 (for example, 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 negation of a conjecture, a process we sometimes refer to as *interning* the (negated) conjecture (meaning, converting it into the data structures used internally by the prover, not putting it in prison).

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. That is, 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 $P \land R$ by introducing the clauses

$$\begin{array}{l} \neg X \lor P \\ \neg X \lor R \\ X \lor \neg P \lor \neg R \end{array}$$

whose conjunction is equivalent to

$$X \Leftrightarrow (P \land R) \quad .$$

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 non-literal 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 1920's in an article by Bibel and Eder [Bibel and Eder 1993].

During the interning process, the prover detects repeated subformulas and represents each occurrence of a repeated subformula by the same proxy, which need be defined only once. The prover 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

$$R \land P$$

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

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 semi-decidable. These observations don't discourage us, since in the ESC application the inputs to the prover are verification conditions, 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 the prover 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 showed that if we turned the simple prover 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 the prover would sometimes get swamped by a combinatorial explosion of case splits when proving a formula of the form $P \wedge Q$, even though it could quickly prove either conjunct individually. Investigation identified the problem that we call "exponential mixing": the prover 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 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, Dill, and Stump [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 non-lazy 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. The prover maintains the invariant that the definition set uniquely specifies all proxy variables in terms of the non-proxy variables. That is if $v_1, \ldots, v_n$ are the non-proxy variables and $p_1, \ldots, p_m$ are the proxy variables, *defs* will be such that the formula

$$\forall v_1, \ldots, v_n : \exists!\, p_1, \ldots, p_m : \mathit{defs} \tag{3}$$

is valid (where ∃! means "exists uniquely"). The context as a whole represents the formula

$$\forall p_1, \ldots, p_m : (\mathit{defs} \Rightarrow (\mathit{cls} \;\wedge\; \mathit{lits}))$$

or equivalently (because of (3))

$$\exists p_1, \ldots, p_m : (\mathit{defs} \;\wedge\; \mathit{cls} \;\wedge\; \mathit{lits}) \quad .$$

Second, we change the way that the prover initializes the context before invoking *Sat*. Specifically, given a query Q, the prover creates proxies for all non-literal subformulas 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

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

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 the prover makes its top-level call to *Sat*.

Third, we slightly modify the specification of *Sat*:

```
proc Sat()
(* Requires that all proxy literals in lits are 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 non-proxy literals in *lits* then

$$(\forall p_1, \ldots, p_m : (\mathit{defs} \;\wedge\; \mathit{cls} \;\wedge\; \mathit{nlits}) \;\Rightarrow\; \mathit{plits}) \quad .$$

Fourth, we modify the implementation of *AssertLit* so that proxy literals are treated specially. When l is a non-proxy 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 list (since they would immediately be subject to clause elimination) and the remaining clauses are width-reduced by deletion of $\neg l$ before being added to the clause list. For example, suppose that s is a proxy for S, that t is a proxy for T, and p is a proxy for $S \wedge T$, so that the definition of p consists of the clauses:

$$\begin{array}{l} \neg p \vee s \\ \neg p \vee t \\ p \vee \neg s \vee \neg t \end{array}$$

Then the call *AssertLit*(p) would add p to *lits* and add the unit clauses s and t to *cls*, while the call *AssertLit*$(\neg p)$ would add the literal $\neg p$ to *lits* and add the clause $\neg s \vee \neg t$ 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. We call this policy of adding the proxy literal to *lits 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 context (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 non-proxy variables.)

The correctness proof for the revised implementation of *Sat* is similar to that for the version of Section 3.2. The important things to show are

(1) that the actions of *Refine*, particularly unit assertion, preserve the meaning of the context,

(2) that when *Sat* performs a case split, the contexts supplied to the recursive calls still represent formulas whose disjunction is equivalent to the formula represented by the parent context (and also, these contexts have no proxy literals in their literal sets), and

(3) that the algorithm maintains the invariant that all proxy literals in *lits* are redundant.

(4) that if *cls* is empty and the proxy literals in *lits* are redundant, then the conjunction of the non-proxy literals in *lits* is equivalent to the entire context.

It is straightforward to show that these things follow from the properties of *defs* described above.

To see how use of lazy CNF can prevent exponential mixing, consider proving a formula 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, non-lazy approach, the initial clause set would contain the unit clause $\neg p_3$; defining clauses for p_3, namely

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

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 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

$$\neg p_1 \vee \neg p_2 \quad ,$$

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 the prover used the subsumption heuristic 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 the prover on the conjecture

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

The prover 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 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.

3.4.2 *Comparison with resolution.* We pause in our exposition to contrast our approach with resolution. The resolution approach relies on the completeness of the rule

$$\frac{\begin{array}{c} P \ \vee \ Q \\ \neg\, P \ \vee \ R \end{array}}{Q \ \vee \ R}$$

for clausal refutation. By applying this rule exhaustively to a set of clauses, the method is certain to refute any unsatisfiable set of clauses. For example, consider using resolution on the clauses produced by the equisatisfiable CNF for the negation of the conjecture

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

from the preceding example. These are:

0. $\neg\, p_3$
1. $p_1 \ \vee \ s$
2. $p_1 \ \vee \ \neg\, t$
3. $\neg\, p_1 \ \vee \ \neg\, s \ \vee \ t$
4. $\neg\, p_2 \ \vee \ s$
5. $\neg\, p_2 \ \vee \ p_1$
6. $p_2 \ \vee \ \neg\, s \ \vee \ \neg\, p_1$
7. $p_3 \ \vee \ p_2$
8. $p_3 \ \vee \ \neg\, t$
9. $\neg\, p_3 \ \vee \ \neg\, p_2 \ \vee \ t$

where clause 0 is the negated proxy for the entire conjecture, clauses 1–3 are the definition of p_1, clauses 4–6 define p_2, and clauses 7–9 define p_3. A refutation is

10. p_2 (inferred from 0 and 7)
11. $\neg\, t$ (from 0 and 8)
12. s (from 4 and 10)
13. p_1 (from 5 and 10)
14. $\neg\, s \ \vee \ t$ (from 3 and 13)
15. $\neg\, s$ (from 11 and 14)
16. **false** (from 12 and 15).

There are three reasons that we use our approach instead of the well-explored method of resolution. First, our method of proof provides a counterexample when the proof fails. Second, the backtracking search incorporates domain-specific algorithms (described in Section 4) into the test for consistency of *lits*. It is not clear how to incorporate domain-specific algorithms into the resolution search. Third, the backtracking search uses less space than resolution.

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, which prepares the input to the theorem prover, 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 VC generator to convey this hint to the theorem prover.

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

```
proc $A()$
  requires $P_1 \vee P_2 \vee \ldots \vee P_m$
  ensures $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n$
```

This will eventually lead to a conjecture of the form

$$P_1 \vee P_2 \vee \ldots \vee P_m \Rightarrow (Q'_1 \wedge \ldots \wedge Q'_n)$$

(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

$$(P_1 \vee P_2 \vee \ldots \vee P_m) \wedge (\neg Q'_1 \vee \ldots \vee \neg Q'_n)$$

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 the theorem prover 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, the prover 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 the prover is lucky enough to split on the relevant clauses first, then the proof search will go quickly. But if the prover 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 the prover to backtrack, the prover 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 be

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

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), if the prover 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,

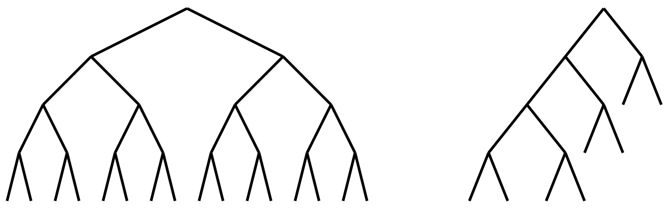


Fig. 1. In one simple case, the scoring heuristic produces the tree at the right instead of the tree at the left.

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. The prover resets scores whenever it backtracks from a case split on a non-goal clause and the previous case split on the current path 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* of D. Suppose the child clause D is relevant and acquires a high score. When the prover 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 the prover 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 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 list. 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.

A subtlety of this scheme is that the prover 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

$(\ldots \lor l \lor \ldots)$ from the clause set, the expansion of l must be added to the clause set at that point. Otherwise the prover might find a "satisfying assignment" 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. (Hint: The top-level call to *Sat* is with a context where *lits* contains no literals, and hence no non-redundant literals, but recursive calls may be made with contexts where *lits* contains some non-redundant proxy literals. What is the right specification for the behavior of *Sat* given such a context?)

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
$AssertLit(L)$
$Push()$
$Pop()$
```

but with L ranging over the literals of Simplify's pre-defined 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 the prover are the *E-graph* module and the 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 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. To achieve this, it is necessary for the two modules to cooperate, according to a protocol 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 [Tinelli and Harandi 1996] and by Nelson [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", 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 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.

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

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

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 $\mathcal{S} \cup \mathcal{T}$, given solutions for $\mathcal{S}$ and $\mathcal{T}$, and assuming that $\mathcal{S}$ and $\mathcal{T}$ both contain the relation symbol $=$ and the fact that it is an equivalence relation, and have no other common function symbols or relation symbols.

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
$f(g_1) \neq f(z)$	$g_1 = g_2 - g_3$
$f(x) = g_2$	$x \leq y$
$f(y) = g_3$	$y + z = x$
	$0 \leq z$

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 paper. The phrase "Nelson-Oppen method" is often used in contrast to the "Shostak method" invented a few years later by Rob Shostak at SRI [Shostack 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, two papers have appeared correcting and clarifying Shostak's method [Rueß and Shankar 2001; Barrett et al. 2002b]. The consensus of these papers seems to be that Shostak's method is not so much an independent combining method but a collection of ideas on how to implement the Nelson-Oppen combination method efficiently. 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:

$$\begin{array}{ll} 4. & f(f(a,b),b) = f(a,b) \quad \text{(from 1, } b = b\text{, and congruence)} \\ 5. & f(a,b) = c \quad \text{(from 4, symmetry on 4, 2, and transitivity)} \\ 6. & a = c \quad \text{(from 5, 1, symmetry on 1, and transitivity)} \\ 7. & g(a) = g(c) \quad \text{(from 6 and congruence)} \\ & \text{which contradicts 3.} \end{array}$$

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. 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 i-th 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 $1 \leq i \leq degree(u)$, $R(u[i], v[i])$. 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*).

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 $node(U)$

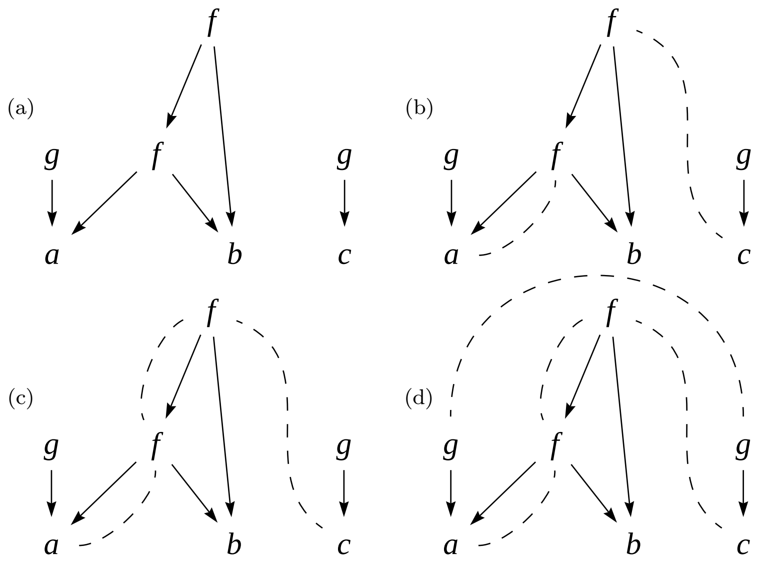


Fig. 2. Application of the congruence closure to example (5). (a) A term DAG for the terms in (5). (b) The E-graph whose equivalences (shown by dashed lines) correspond to the equalities in (5). (c) $\mathit{node}(f(f(a,b),b))$ and $\mathit{node}(f(a,b))$ are congruent in (b); make them equivalent. (d) $\mathit{node}(g(a))$ and $\mathit{node}(g(c))$ are congruent in (c); make them equivalent. Since $g(a)$ and $g(c)$ are distinguished in (5) but equivalent in (d), (5) is unsatisfiable.

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 in the last paragraph 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 as symbolic literal constants. The E-graph pictured in Figure 2 makes them labeled nodes with zero arguments, i.e., 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 $\mathit{node}(T_1)$ and $\mathit{node}(T_2)$ 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 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, 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, Sethi, and Tarjan [Downey et al. 1980], our implementation guarantees that incrementally asserting the literals of any 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 numeric inequality 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 9.14).

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 the prover 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, the prover includes an incremental, resettable decision procedure for satisfiability of conjunctions of T-literals. 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 the prover. 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 $T.\mathit{Unknown}$ 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 $T.\mathit{Unknown}$ (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 $T.\mathit{Unknown}$ whatever fields its satisfiability procedure needs, but there is one field common to all the various $T.\mathit{Unknown}$ 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 $e.T\_\mathit{unknown} \neq \mathbf{nil}$ then $e.T\_\mathit{unknown}.\mathit{enode}$ is equivalent to e, and

(2) for any $T.\mathit{Unknown}\ u$, if $u.\mathit{enode} \neq \mathbf{nil}$ then $u.\mathit{enode}.T\_\mathit{unknown} = u$.

An unknown u is *connected* if $u.\mathit{enode} \neq \mathbf{nil}$.

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

```
proc $T.\mathit{UnknownForEnode}(e : \mathit{Enode}) : \mathit{Unknown}$;
(* Requires that $e.\mathit{root} = e$. Returns $e.T\_\mathit{unknown}$ if $e.T\_\mathit{unknown} \neq$
nil. Otherwise sets $e.T\_\mathit{unknown}$ to a newly-allocated unconstrained un-
known (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:

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

where the u's are unknowns of T. There must be procedures for building these literals from unknowns, but we won't 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 will remove and assert 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 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). 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, Section 13].

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 `AF`'s as much as possible, so that, for example, if 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, but Simplify leaves it to each ordinary theory to expend an appropriate amount of effort in canonicalizing applications of its own relation symbols.

In Simplify, in our determination to distinguish functions from relations, we did not make `AF` a subtype of *Enode*: 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) 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 $\mathbf{neq}(t, u) = @\mathbf{true}$ 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 deletion. 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 consistent literal sets containing only uninterpreted propositional variables and their negations, as in Section 3, containment ($l \in lits$) and implication ($[lits \Rightarrow l]$) 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 $[lits \Rightarrow l]$. *)


proc $Implied(l) \equiv$
  $Push()$;
  $AssertLit(\neg l)$;
  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 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.

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, 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 not by either of the status test or the E-graph test.

In an early version of the prover 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 the prover, we apply the plunging version of *Refine* to each non-unit clause produced by matching (see Section 5), 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 quasi-relations, the syntax

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

is somewhat like a higher order atomic formula that asserts that f and g are the irreflexive and reflexive versions 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 won't describe all its details in this paper. 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

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

from

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

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, semi-decidable: 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 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.

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, the prover 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.

The part of the theorem prover 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, the prover 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

$$(\forall x, y : \mathit{car}(\mathit{cons}(x, y)) = x) \qquad .$$

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

$$\mathit{cons}(a, b) = \mathit{cons}(c, d) \Rightarrow a = c \qquad ,$$

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 matching rule A might trigger a new instance of a matching rule 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 5.1, and the trigger selection "loop test" of Section 5.2. 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 5.1.

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

$$(\forall s, t, x : \mathit{member}(x, s) \ \wedge \ \mathit{subset}(s, t) \ \Rightarrow \ \mathit{member}(x, t)) \qquad ,$$

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

When we say that an instance $\theta(t)$ "is represented" in an E-graph, we mean that it is represented up to equivalence. That is, we mean that the E-graph contains an E-node whose value must be equal to $\theta(t)$, given the equalities in the graph. 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.

Because an E-graph represents more terms than it has E-nodes, 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

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

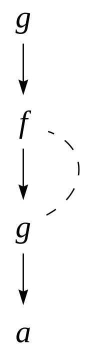
follows from

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

(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 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.



$$\begin{tikzcd} g \arrow[d] \\ f \arrow[d] \arrow[d, dashed, bend left, no head] \\ g \arrow[d] \\ a \end{tikzcd}$$

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 (See "Complexity of finitely generated algebras. Dexter Kozen. Proceedings of the ninth Symposium on Theory of Computing. pp 164–77.) 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 the prover 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 ground E-graph would contain many more ground terms 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.

The prover transforms quantified formulas into internal data structures called *matching rules*. A *matching rule* mr is a triple consisting of a *body*, written $mr.body$, which is a formula; a *variable-list*, written $mr.vars$, which is a list of variables; and a *trigger-list*, written $mr.triggers$, which is a list of *triggers*, each of which is a list of one or more terms.

The prover 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, the prover will use only those instances $\theta(mr.body)$ of the matching rule such that for some trigger tr in $mr.triggers$, for each term t in the list tr, $\theta(t)$ is represented in the E-graph.

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

—Searching and matching: 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.

5.1 Searching and matching

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 5.2.

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 5.3.

The high-level description of the interaction of searching and matching is very simple: periodically during the backtracking search, the prover 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 the prover has a choice between matching and case splitting, it favors matching.

Second, the prover 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. 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 the prover will to some extent alternate between case-splitting and matching, and therefore avoids matching loops. Disabling the activation has a disastrous effect on performance (see Section 9.7). 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 later in this section.

Third, the first time a ground clause is created as an instance of a matching rule, we find it worthwhile to refine it aggressively, by plunging. If a literal is found to be untenable by plunging, it is deleted from the clause and also explicitly denied (a kind of subsumption heuristic). 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, the prover 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 finds a ground 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 the prover 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 that this 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.

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 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* (described in Section 3.2). Recall that the purpose of *Refine* is to perform tactics, such as width reduction and unit propagation, 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;
        elsif refinement enabled then
          for each clause $C$ in cls do
            refine $C$ by cheap tests (possibly setting refuted)
          end;
          disable refinement
        elsif matching enabled then
          for each asserted matching rule $M$ do
            for each substitution $\theta$
              that matches some trigger in $M.triggers$ to the E-graph do
              let $fp = fingerprint((M, \theta))$ in
                if not $fp \in matchfp$ then
                  add $fp$ to $matchfp$;
                  let $C = \theta(M.body)$ in
                    refine $C$ by plunging (possibly setting refuted);
                    add $C$ to cls
                  end
                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 the prover'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 non-unit 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.

5.1.1 *The matching depth heuristic.*. Matching also affects the choice of which case split to perform.

We have mentioned the goal bit and the score 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, 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&goal ties by favoring high scores.

Implementation note: At any moment, the prover is considering clauses only of the current depth and the next higher depth. Therefore, the prover 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, the prover 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 list, 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:

$$(\forall a, i, x, j : i = j \ \vee \ \mathit{select}(\mathit{store}(a, i, x), j) = \mathit{select}(a, j))$$

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 non-unit 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 $\mathit{select}(\mathit{store}(a, i, x), j))$ even in the inactive portion of the E-graph. For each such instance, the tactic uses the E-graph test (Section 4.6) to test 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 $\mathit{select}(a, j)$ 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 9.5 and 9.6). 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.

5.2 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.

5.2.1 *Simple story.* We define a *basic literal* to be a non-proxy literal.

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

$$\begin{aligned} P \Leftrightarrow Q &= (P \Rightarrow Q) \ \wedge \ (Q \Rightarrow P) \\ P \Rightarrow Q &= (\neg P \ \vee \ Q) \end{aligned}$$

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

$$\begin{aligned} \neg (P \ \wedge \ Q) &= (\neg P) \ \vee \ (\neg Q) \\ \neg (P \ \vee \ Q) &= (\neg P) \ \wedge \ (\neg Q) \\ \neg ((\forall x : P)) &= (\exists x : \neg P) \\ \neg ((\exists x : P)) &= (\forall x : \neg P) \\ \neg \neg P &= P \end{aligned}$$

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.*

All the elimination rules are applied together in one pass over the formula. 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 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 refine instances of the rule, as described previously (Section 3.2). 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

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

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

The prover'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 the prover makes two tries to select triggers automatically.

First try: the prover makes a uni-trigger 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 the prover 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. The prover's 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 uni-triggers, then the prover tries to create a multi-trigger. It attempts to select as the constituents a reasonably small set of non-proscribed 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 multi-trigger. Because there could be exponentially many plausible multi-triggers, the prover 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 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.

5.2.2 *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 context without ever asserting the formula at all. Therefore, the prover 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 backtracking search begins. However, universal quantifiers that are not outermost, but are nested within other universal quantifiers, are simply treated as opaque 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 semi-normal 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 semi-normal 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 semi-normal 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 *non-clausal 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, the prover estimates the size of the result. If the estimate is enormous, it produces a matching rule whose body is the unnormalized quantifier body. The trigger for such a rule is computed from the atomic formulas in the body just as if these atomic formulas were the elements of a clause. When a non-clausal rule is instantiated, the instantiation of its body is asserted and treated by the equisatisfiable CNF methods described previously.

Recall that the prover favors case splits with low matching depth. This heuristic prevents the prover 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. Non-clausal 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; the prover must not let itself be tricked into asserting this proxy prematurely.

Finally, the prover's policy of treating nested quantifiers as opaque subformulas can cause trigger selection to fail. For example, in the formula

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

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

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

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

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

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 the prover; on the other hand, it is important that the input language be expressive enough to direct the prover towards heuristically desirable search strategies.

5.3 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 $\theta$
    that matches some trigger in $M.triggers$ to the E-graph do
    ...
  end
end
```

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

5.3.1 *Matching iterators.* We present the matching algorithms as mutually recursive iterators in the style of CLU. 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.

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 $MatchTrigger(t : \text{list of proper terms}, \theta : \text{substitution})$
   yields all extension $\theta \cup \phi$ of $\theta$ such that
   $\theta \cup \phi$ matches the trigger $t$ in the E-graph and
   $\phi$ does not conflict with $\theta$.

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

iterator $Match(t : \text{term}, v : Enode, \theta : \text{substitution})$
    yields all extensions $\theta \cup \phi$ of $\theta$ such that
    $\theta \cup \phi$ matches $t$ to $v$, and
    $\phi$ does not conflict with $\theta$.

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

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 $\theta$ 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 $MatchTrigger(t : \text{list of proper terms}, \theta : \text{substitution}) \equiv$
   if $t$ is empty then
      yield($\theta$)
   else
      for each $\phi$ in $MatchTerm(\text{hd}(t), \theta)$ do
         for each $\psi$ in $MatchTrigger(\text{tl}(t), \phi)$ do
            yield($\psi$)
         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 $MatchTerm(t : \text{proper term}, \theta : \text{substitution}) \equiv$
  let $f$, $args$ be such that $t = f(args)$ in
    for each active E-node $v$ labeled $f$ do
      for each $\phi$ in $MatchList(args, children[v], \theta)$ do
        yield($\phi$)
      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 $MatchList(t : \text{list of terms}, v : \text{list of } Enode\text{'s}, \theta : \text{substitution}) \equiv$
  if $t$ is the empty list then
    yield($\theta$)
  else
    for each $\phi$ in $Match(\text{hd}(t), \text{hd}(v), \theta)$ do
      for each $\psi$ in $MatchList(\text{tl}(t), \text{tl}(v), \phi)$ do
        yield($\psi$)
      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, $v$ : Enode, $\theta$ : substitution) ≡
  if $t$ is a pattern variable then
    if $t$ is not in the domain of $\theta$ then
      yield($\theta \cup \{(t, v$'s equivalence class$)\}$)
    elsif $\theta(t)$ contains $v$ then
      yield($\theta$)
    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 $\phi$ in MatchList(args, children[$u$], $\theta$) do yield($\phi$) end
      end
    end
  end
end
```

We conclude this subsection 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, 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.

The prover 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 $\theta(t)$, yields θ if it does, and yields nothing if it doesn't.

When Simplify constructs a trigger 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 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.

5.3.2 *The mod-time matching optimization.* The prover spends a considerable fraction 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 *pair-set* optimization.

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

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 $\theta$ such that $\theta(T)$ is equivalent to $V$ do
      $Assert(\theta(B))$
    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. 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. 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 $\theta$ such that $\theta(T)$ is represented in the E-graph,
        either $\theta(B)$ has already been asserted, or
        the E-node $V$ that represents $\theta(T)$ satisfies $V.mt = gmt$
```

Given the mod-time invariant, the our 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 $\theta$ such that $\theta(T)$ is equivalent to $V$ do
```

```
      $Assert(\theta(B))$
    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 assumed. The reason that $gmt := gmt + 1$ maintains the invariant is that after a matching round, all instances of rules that match have been assumed.

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 $UpdateMT(V : Enode) \equiv$
  for each $P$ such that $P$ is a parent of some E-node equivalent to $V$ do
    if $P.mt < gmt$ then
      $P.mt := gmt$;
      $UpdateMT(P)$
    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.

All of these updates to the *mt* fields of E-nodes must be undone by *Pop*.

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 multi-triggers. This is an awkward problem. Consider the case of a multi-trigger p_1, ..., p_n, and suppose that merges change the E-graph so that there is a new instance θ of the multi-trigger; 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 multi-trigger 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 trigger term 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 trigger terms 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 maintain a boolean property of matching rules, *newness*, and weaken the mod-time invariant by limiting its outer quantification to apply to all non-new matching rules, instead of to all matching rules. During a round of matching, new rules get 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 $T_1, \dots, T_n$ and body $B$
  either $mr.new$ or
  **for all** substitutions $\theta$ such that each $\theta(T_i)$ is represented
          by an active E-node in the E-graph,
              either $\theta(B)$ has already been asserted, or
              there exists an $i$ such that the E-node $V$ that represents
              $\theta(T_i)$ 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 $\theta$ such that $(\forall t \in tr : \theta(t)$exists and is active$)$ do
        $Assert(\theta(mr.body))$
      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 $\phi$ such that $\phi(t)$ is equivalent to $V$ do
            for each extension $\theta$ of $\phi$ such that
                $(\forall q \in tr : \theta(q)$exists and is active$)$ do
              $Assert(\theta(mr.body))$
            end
          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 (and by undoing this in *Pop*).

5.3.3 *The pattern-element matching optimization.* 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 a 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

$$f(\dots,\ g(\dots),\ \dots) \quad ,$$

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

$$f(\dots,\ x,\ \dots) \quad ,$$

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

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

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 $\theta$ 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 (and undo the additions when merges are undone).

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, the prover

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

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

(3) adds a pair (f, g) to *gpp* for each g 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 multi-trigger, any trigger term 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 multi-trigger one of whose terms is an application of f to distinct pattern variables that do not occur elsewhere in the multi-trigger.

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*.

5.3.4 *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 [1..64] (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 Q in the E-graph, we associate two approximate sets of function symbols: $lbls(Q)$ and $plbls(Q)$, where $lbls(Q)$ is the set of labels of active E-nodes in Q and $plbls(Q)$ is the set of labels of active parents of E-nodes in Q. The code for merging two equivalence classes, say Q into R, is extended with the following:

$$gpc := gpc \cup plbls(Q) \times lbls(R) \cup lbls(Q) \times plbls(R);$$

```
$gpp := gpp \cup plbls(Q) \times plbls(R)$;
Save $lbls(R)$ on the undo stack;
$lbls(R) := lbls(R) \cup lbls(Q)$;
Save $plbls(R)$ on the undo stack;
$plbls(R) := plbls(R) \cup plbls(Q)$;
```

The operations on $lbls(R)$ and $plbls(R)$ must be undone by *Pop*. This requires saving the old values of the approximate sets on the undo stack. Since the prover 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. 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 pairs 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 pair (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)`.

5.3.5 *Mod-times and pattern elements and multi-triggers.* There is one final optimization to be described that uses pattern elements and mod-times. Recall that when matching a multi-trigger, each term in the multi-trigger 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 terms in the multi-trigger, without regard to mod-times). The need to consider each term in the multi-trigger 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 multi-trigger.

As a simple example, consider the multi-trigger

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

An E-graph contains an instance of this multi-trigger 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 multi-trigger is considered for matching, *gpc* $= \{(f, g)\}$ and *gpp* is empty. For any new instance (U, V, W) of the multi-trigger, 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 multi-trigger, 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 multi-trigger, and suppose that when the multi-trigger 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 multi-trigger 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 multi-trigger, and suppose that when the multi-trigger 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 multi-trigger with trigger terms $p_1, \ldots, p_n$, 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 pair 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 pair for x_i, the set of gating terms includes some term 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 pairs for v_j or some parent-parent pair 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 *depth* 1 if it has no parent-child elements. We define a multi-trigger to be *tree-like* if none of its pattern variables occurs more than once within it. We define the *shape* of a multi-trigger T as the tree-like multi-trigger obtained from T be replacing the i-th 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 multi-trigger 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 multi-trigger 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 multi-trigger, then it must have created a match of some constituent of that trigger. There are two subcases. If the constituent is of depth greater than 1, then the enabling modification must have added to *gpc* a parent-child element of some constituent of the multi trigger, 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 depth 1, 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 multi-trigger, it must have added some parent-parent pair 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, the prover 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 test detects the redundancy, so the unnecessary clauses are not added to the clause set.

6. REPORTING ERRORS

Many conjectures are not theorems. Very frequently the theorem prover's backtracking search finds a context that falsifies the conjecture. In this section, we call such a context an *error context*, a more precise term than "counterexample".

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.

Simplify uses two methods to present the reason for invalidity of the conjecture: *error context reporting*, and *error localization.* Error context reporting describes the context in which the conjecture is false. Error localization attributes the invalidity of the conjecture to a specific piece of its text.

Each method must be implemented to avoid the large degree of arbitrariness in the high-level description: a given error context can be reported in many semantically equivalent but syntactically different ways; and the invalidity of an implication could be attributed either to the weakness of its antecedent or to the strength of its consequent.

6.1 Error context reporting

As an example of error context reporting, the invalidity of the conjecture $x \geq 0 \Rightarrow x > 10$, would be reported with the context $\{x \geq 0, x \leq 10\}$.

When the backtracking search finds a context that falsifies the conjecture, we would like to produce a compact textual description of the context. Producing a compact textual description of a context from the Simplify data structures that represent it—the *context printing problem*—is a difficult one. In the special case of the E-graph, there is an efficient and elegant algorithm for constructing a minimum-sized set of equalities whose congruence closure is a given E-graph [Nelson 1981, Section 11]. Simplify uses this algorithm to print the E-graph portion of the error context. Simplify also applies a similar, but less elegant, algorithm to the Simplex tableau to produce a minimal set of linear inequalities that describe the arithmetic portion of the error context.

One difficulty with the context printing problem relates to Simplify's *background predicate*, which we now describe. The background predicate is an arbitrary formula that is given once and then used as an implicit antecedent for a number of different conjectures. 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 of the procedures in the module one by one.

The background predicate exacerbates the difficulties of the context printing problem: the aforementioned "minimal" representations of the E-graph and the Simplex tableau may include 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 rejecting 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, having implemented error localization, we sometimes use the switch that disables error context reporting altogether.

A fine point: We never aspired to include quantified formulas in the reported error context, but only basic literals.

6.2 Error localization

For most Simplify applications, including ESC, error localization is preferable to error context reporting. The fundamental reason is that error context reporting doesn't direct the user's attention to the textual source of the error. Even after pruning counterexamples are often large. 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 $i \geq 0$ (page 10, line 30)". When Simplify is used by ESC, we would like to know the source location and nature of the program error that led to the invalid verification condition.

To cope with the arbitrariness of error localization, 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 the theorem prover 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 the prover 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. (The prover 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

$$\textit{Precondition} \Rightarrow p \neq \textbf{null} \ \wedge \ i \geq 0 \ \wedge \ \ldots \quad .$$

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

$$\begin{array}{l} \textit{Precondition} \Rightarrow \\ \quad |\texttt{Null@10}|^* : p \neq \textbf{null} \ \wedge \ |\texttt{IndexNegative@11}|^* : i \geq 0 \ \wedge \ \ldots \quad . \end{array}$$

Thus if the proof fails, the prover 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 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

$$\begin{array}{l} \textit{Precondition} \Rightarrow \\ \quad (|\texttt{EV.Null@10}| \ \vee \ p \neq \textbf{null}) \ \wedge \ (|\texttt{EV.IndexNegative@11}| \ \vee \ i \geq 0) \ \wedge \ \ldots \quad . \end{array}$$

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 by transforming atomic formulas into non-atomic 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, the prover keeps labels attached to occurrences of literals (including proxy literals), and when an error context is discovered, the positive labels of the asserted literals and the negative labels of the denied literals are printed.

By setting an optional switch, the prover can be directed to print a log of each label as the search encounters it. This produces a dynamic trace of which proof obligation the prover 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. Of the many details, we will mention three.

First, the prover sometimes has to rewrite the propositional structure of a conjecture; for example when transforming the body of a universally quantified formula into CNF. The rewriting rules must take account of the labels. For example, we use the rule

$$P \ \wedge \ L : \mathbf{true} \longrightarrow L : P \quad .$$

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

$$\begin{aligned} P \ \vee \ L : \mathbf{true} &\longrightarrow L : \mathbf{true} \\ P \ \vee \ L^* : \mathbf{false} &\longrightarrow L^* : P \\ P \ \wedge \ L^* : \mathbf{false} &\longrightarrow L^* : \mathbf{false} \quad . \end{aligned}$$

More problematically, we may need to put a formula like $P \ \vee \ (Q \ \wedge \ R)$ into CNF by distributing $\vee$ into $\wedge$. If the conjunction is labeled, we have a problem. To solve this problem, suppose that for each label L, we can construct two "virtual labels" $L\alpha$ and $L\beta$. If both $L\alpha$ and $L\beta$ are output by the prover, then L is output to the user, but if only one of them is output, it is suppressed and the user sees nothing. With the aid of this new mechanism, we have:

$$\begin{aligned} P \ \vee \ L : (Q \ \wedge \ R) &\longrightarrow (P \ \vee \ L\alpha : Q) \ \wedge \ (P \ \vee \ L\beta : R) \\ P \ \vee \ L^* : (Q \ \wedge \ R) &\longrightarrow (P \ \vee \ L^* : Q) \ \wedge \ (P \ \vee \ L^* : R) \end{aligned}$$

Second, it is possible to label formulas that are inside the scope of a universal quantifier, and we must distinguish the possibly many occurrences of the label in distinct instances of the quantified formula. We therefore allow labels to have parameters, which are terms. When the formula $(\forall x : P)$ is instantiated to produce the instance $P(x := E)$, any label L in P is changed to $L(E)$. Labels within quantifiers have not been common when the prover was used for ESC. We have used the prover for a few other purposes, which suggest to us that the approach of tagging these labels with parameters could be useful, but our experience is not extensive enough to justify high confidence.

There is an interaction between the virtual labels $L\alpha$ and $L\beta$ that were introduced for the rewriting into CNF, and the label parameters introduced for labels within quantifiers. Basically, if $L(E)\alpha$ and $L(F)\beta$ are both output by the prover, then no label is output to the user if E and F are different terms; if they are the same term, then $L(E)$ is output.

Third and finally, labels interact with canonicalization. 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 that $(M : P) \Rightarrow Q$. We don't have any data to know whether this is hurting performance.

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 the prover 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, Sethi, and Tarjan [Downey et al. 1980], but this section describes the module in more detail, including propagating equalities, handling distinctions, and undoing merges. First we introduce the features and invariants of the basic E-graph data structure, and then we give pseudo-code 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 equality between the corresponding terms is a logical consequence of the current equality assertions.

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 $p, p.next, p.next.next, \ldots$ 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 re-root the smaller of the two equivalence classes. Although the union-find algorithm analyzed by Tarjan [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 term DAG, in which nodes are labeled and can have any natural number of out-edges, we use a data structure that we call an 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 of a DAG node by a list linked by the *cdr* fields. Here is a precise statement of this basic idea:

Definition. 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 non-empty 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.

In the binary 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. The definitions of *congruence-closed* and *congruence closure* are built on the definition of *congruent* exactly as in the definitions in Section 4.2.

In the body of the paper, we used the terms *E-graph* and *E-node* to refer to the abstract term DAG and its nodes, but in this section we will be careful to use these terms only for the representation E-graph and its binary and atomic nodes.

We leave it to the reader to verify that if Q is the congruence closure in the new sense of the relation R on term nodes, then Q restricted to term nodes is the congruence closure of R in the sense of Section 4.2.

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. (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.

Returning to the implementation of congruence closure, the reduction to the binary 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 now describe how the parents of an equivalence class are enumerated.

We define 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. (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.

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
      $r.parent, r.parent.samecar, r.parent.samecar.samecar, \ldots$ do
      Visit $p$
    end
  else
    for each node $p$ in the circular list
      $r.parent, r.parent.samecdr, r.parent.samecdr.samecdr, \ldots$ 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*, which are atomic formulas of the form $X \neq Y$. 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. Specifically, there are two fields used to represent unmergeability:

—Each root r contains a list *r.forbid* of nodes that are unmergeable with r.

—Each root r contains a set *r.distClasses*, represented as a bit vector, and two distinct roots r and s are unmergeable if *r.distClasses* and *s.distClasses* have a nonempty intersection.

To assert $X \neq Y$, 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*. By extending both forbid lists, we ensure that (up to equivalence) x is on y's forbid list if and only if y is on x's. That is, we maintain

Invariant. (FORBID LIST SYMMETRY) For any two E-node roots x and y, *x.forbid* contains a node equivalent to y if and only if *y.forbid* contains a node equivalent to x.

This invariant speeds up the test for inconsistency when two equivalence class roots are merged, since the invariant makes it sufficient to traverse the shorter of the two forbid 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.

In addition to binary distinctions like $X \neq Y$, Simplify accepts the more general formula

$$DISTINCT(t_1, \ldots, t_n), \tag{10}$$

which asserts that all the t's are distinct. To express this with $\neq$ alone would require a conjunction of length $O(n^2)$. The *distClasses* field allows us to handle *DISTINCT* efficiently: To assert the formula (10), the prover first checks whether any of the t's are already equivalent, in which case it detects a contradiction. Otherwise, it allocates a new bit position and sets that bit in the *distClasses* field of the nodes $t_1.root, \ldots, t_n.root$. Our prover uses this technique for up to k occurrences of *DISTINCT*, where k is the number of bits per word in the host computer. For any additional occurrences of *DISTINCT*, we retreat 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 sixty-four bit Alpha processors, the retreat never became a problem.

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

```
proc Unmergeable(x, y : Enode) : 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, Section 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 a root r, $r.cgPtr = r$. For a non-root x, $x.cgPtr$ is the tree parent of x.

A *congruence root* is an E-node x such that $x = x.cgPtr$. In Section 5.3, 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 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*

The equality $X = Y$ is asserted by the call $AssertEQ(x, y)$ 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 equivalence 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*. The call $Merge(x, y)$ combines the equivalence classes of x and y, adding pairs of nodes to *pending* as new congruences are detected.

```
proc $AssertEQ(x, y : Enode) \equiv$
  $pending := \{(x, y)\}$;
  while $pending \neq \{\}$ do
    remove a pair $(p, q)$ from $pending$;
    $p, q := p.root, q.root$;
```

```
      if $p \neq q$ then
         (* $p$ and $q$ are not equivalent. *)
         if $Unmergeable(p, q)$ then
            $refuted := \textbf{true}$;
            return
         else
            $Merge(p, q)$
         end
      end
   end
   (* The E-graph is congruence-closed. *)
end
```

Merge requires that its arguments be two E-nodes x and y representing terms X and Y such that $X = Y$ is implicit in the input, but neither $X = Y$ nor $X \neq Y$ is already represented in the E-graph.

We now present pseudo-code for *Merge*.

```
proc $Merge(x, y : Enode) \equiv$
```

The implementation of $Merge(x, y)$ begins, like many union-find algorithms, by swapping x and y if necessary to make x the root of the larger class, which will become the root of the combined class.

```
   (* Make $x$ be the root of larger equivalence class. *)
   if $x.size < y.size$ then
      $x, y := y, x$
   end;
```

Next, we propagate equalities to the ordinary theories and ensure that the new root x references an unknown if any node in the merged equivalence class does.

```
   for each ordinary theory $T$ do
      (* maintain PROPAGATION TO $T$ *)
      if $x.T\_unknown \neq$ nil and $y.T\_unknown \neq$ nil then
         Propagate the equality of $x.T\_unknown$ with $y.T\_unknown$ to $T$
      end
      (* maintain ROOT $T\_unknown$ *)
      if $x.T\_unknown =$ nil then $x.T\_unknown := y.T\_unknown$ end;
   end;
```

The next piece of code maintains data structures used by the matching optimizations described in Section 5.3.3.

```
   (* Maintain sets for pattern-element optimizations. *)
   Update $gpc$, $gpp$, $lbls(x)$ and $plbls(x)$ as described in Section 5.3.3.
```

Next, we make sure that the root x of the merged class is unmergeable with precisely those E-nodes that were formerly unmergeable either with x or with y.

```
(* Merge forbid lists. *)
**if** $y.forbid \neq$ **nil then**
   **if** $x.forbid =$ **nil then**
      $x.forbid := y.forbid$
   **else**
      $x.forbid.link, y.forbid.link := y.forbid.link, x.forbid.link$
   **end**
**end**;
(* Merge distinction classes. *)
$x.distClasses := x.distClasses \cup y.distClasses$;
```

The union of the *distClasses* fields is implemented as a bitwise OR operation on machine words.

Next, we update *sigTab*, adding pairs of nodes to *pending* as new congruences are discovered.

First we introduce a local variable w to be the root of the class with fewer parents.

```
**var** $w$ **in**
   (* Let $w$ be the class with fewer parents. *)
   **if** $x.parentSize < y.parentSize$ **then**
      $w := x$
   **else**
      $w := y$
   **end**;
```

The parents of w's class are the nodes whose signatures will change as a result of this merge. To keep the signature table up to date, we first remove their old signatures.

```
   (* Remove old signatures. *)
   **for each** parent $p$ of $w$'s equivalence class **do**
      **if** $p$ is a congruence root **then**
         Remove $((p.car.root.id, p.cdr.root.id), p)$ from the signature table
      **end**;
```

Earlier in this section, we described how to implement the **for each** in the code above.

Next we actually perform the union operation on the equivalence classes by making all elements of y's equivalence class point to the new root x. We then splice the circular *next* lists and update $x.size$.

```
   (* Union the equivalence classes. *)
   **for each** $v$ in the circular list $y, y.next, y.next.next, \ldots$ **do**
```

```
    $v.root := x$
  **end**;
  $x.next, y.next := y.next, x.next$;
  $x.size := x.size + y.size$;
```

To ensure that it will be the parents of w's class whose signatures change, we swap the *id* fields of x and y if necessary.

```
  (* Preserve signatures of larger parent set. *)
  **if** $x.parentSize < y.parentSize$ **then**
    $x.id, y.id := y.id, x.id$
  **end**;
```

We now reinsert the nodes with their new signatures. A parent p of w's equivalence class is a congruence root before the merge precisely if $p = p.cgPtr$. If such a congruence root participates in a new congruence, we modify its *cgPtr* to reference the root of the congruence class it joins and we propagate the congruence to the pending list. Old congruence roots that do not participate in new congruences will be congruence roots after the merge, and we insert them into the signature table.

```
  (* Insert new signatures and propagate congruences. *)
  **for each** parent $p$ of $w$'s equivalence class **do**
    **if** $p = p.cgPtr$ **then**
      **if** the signature table contains an entry $q$ under the key
          $(p.car.root.id, p.cdr.root.id)$ **then**
        $p.cgPtr := q$;
        $pending := pending \cup \{(p, q)\}$
      **else**
        Insert $((p.car.root.id, p.cdr.root.id), p)$ into the signature table
      **end**
    **end**
  **end**
**end**;
```

Next we update the data structures that represent the set of parents of the merged equivalence class.

```
(* Merge parent lists. *)
**if** $y.parent \neq$ **nil then**
  **if** $x.parent =$ **nil then**
    $x.parent := y.parent$
  **else**
    (* Splice the parent lists of $x$ and $y$. *)
    **if** $x$ and $y$ are term nodes **then**
      $x.parent.samecar, y.parent.samecar :=$
          $y.parent.samecar, x.parent.samecar$
    **else**
```

```
            $x.parent.samecdr, y.parent.samecdr :=$
                $y.parent.samecdr, x.parent.samecdr$
          end
        end
      end;
      $x.parentSize := x.parentSize + y.parentSize$;
```

The next piece of code maintains the data needed for the mod-time matching optimization described in Section 5.3.2

```
      (* Update mod-times of $x$'s ancestors. *)
      Update mod-times of $x$'s ancestors as described in Section 5.3.2.
```

Finally, we push a record containing the information needed by *Pop* to undo the merge.

```
      (* Push undo record. *)
      Push onto the undo stack the pair ("UndoMerge", $y$);
    end
```

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). We derive the body of *UndoMerge* piece by piece, working backwards through the *Merge* procedure.

We refer to the execution of *Merge* that pushed the undo record (`"UndoMerge"`, y) as the *pertinent merge*. *UndoMerge* starts with the E-graph state that existed after the pertinent merge and restores the E-graph to the state it had before the pertinent merge.

Given y, we compute the other argument x to the pertinent merge:

```
      $x := y.root$
```

We do not describe the undoing of

```
      (* Update mod-times of $x$'s ancestors. *)
      Update mod-times of $x$'s ancestors as described in Section 5.3.2.
```

here. The mod-time updating code creates its own undo records. Since the *UndoMerge* code is independent of the mod-time undoing (and vice versa), it doesn't matter that these undo records are processed out of order with the rest of *UndoMerge*.

The next piece of *Merge* to undo is

```
      (* Merge parent lists. *)
      if $y.parent \neq$ nil then
        if $x.parent =$ nil then
        $x.parent := y.parent$
      **else**
        (* Splice the parent lists of $x$ and $y$. *)
        **if** $x$ and $y$ are term nodes **then**
          $x.parent.samecar, y.parent.samecar :=$
              $y.parent.samecar, x.parent.samecar$
        **else**
          $x.parent.samecdr, y.parent.samecdr :=$
              $y.parent.samecdr, x.parent.samecdr$
        **end**
      **end**
    **end**;
    $x.parentSize := x.parentSize + y.parentSize$
```

Before the pertinent merge, x and y were both roots. Also, they are either both term nodes or both list nodes. If x and y are term [list] nodes, $x.parent.car$ [$x.parent.cdr$] must have been equivalent to x and $y.parent.car$ [$y.parent.cdr$] must have been equivalent to y. But x and y were not equivalent to each other. Thus $x.parent$ and $y.parent$ could not have been equal. So we have $x.parent = y.parent$ after the pertinent merge if and only if $x.parent$ was **nil** before the pertinent merge. So the following code undoes the effect of the code above.

```
    $x.parentSize := x.parentSize - y.parentSize$;
    **if** $y.parent \neq$ **nil then**
      **if** $x.parent = y.parent$ **then**
        $x.parent :=$ **nil**
      **else**
        (* Unsplice the parent lists of $x$ and $y$. *)
        **if** $x$ and $y$ are term nodes **then**
          $x.parent.samecar, y.parent.samecar :=$
              $y.parent.samecar, x.parent.samecar$
        **else**
          $x.parent.samecdr, y.parent.samecdr :=$
              $y.parent.samecdr, x.parent.samecdr$
        **end**
      **end**
    **end**
```

The next fragment of *Merge* to consider is somewhat longer and its inversion is more subtle. It has the following structure:

```
    **var** $w$ **in**
      (* Let $w$ be the class with fewer parents. *)
      ...
      (* Remove old signatures. *)
      ...
      (* Union the equivalence classes. *)
      ...
      (* Preserve signatures of larger parent set. *)
      ...
      (* Insert new signatures and propagate congruences. *)
      ...
    end;
```

We will invert the parts of this block in reverse order, with two exceptions: (1) We begin by computing w, since it is needed in the inverses of last four parts. (2) The inverse of "Insert new signatures and propagate congruences" is partially postponed, the postponed part being performed during the inversion of "Remove old signatures".

Having already restored *x.parentSize* to the value it had before the pertinent merge, we compute w as follows:

```
var w in
  if x.parentSize < y.parentSize then
    w := x.parent
  else
    w := y.parent
  end;
```

With w initialized, we are now ready to invert the last part of the forward code in w's scope. Recall that this code is:

```
(* Insert new signatures and propagate congruences. *)
for each parent p of w's equivalence class do
  if $p = p.cgPtr$ then
    if the signature table contains an entry q under the key
        $(p.car.root.id, p.cdr.root.id)$ then
      $p.cgPtr := q$;
      $pending := pending \cup \{(p, q)\}$
    else
      Insert $((p.car.root.id, p.cdr.root.id), p)$ into the signature table
    end
  end
end
```

This code examines all parents whose signatures change during the pertinent merge. For each parent p that was a congruence root before the pertinent merge:

1. If p does not participate in any new congruences, then p is simply entered into the signature table under its new signature.
2. If p participates in a new congruence, then
   2a. a new equality is added to the the work list *pending*, and
   2b. *p.cgPtr* is set to the root of the congruence class which it has joined.

The effect 1 is undone by the following fragment of the unmerging code. (The effects of merges resulting from 2a will have been undone by the time we start undoing the pertinent merge; we postpone undoing the effects 2b until later in *UndoMerge*.)

```
for each parent $p$ of $w$'s equivalence class do
  if $p$ is a congruence root then
    Remove $((p.car.root.id, p.cdr.root.id), p)$ from the signature table
  end;
end;
```

Next, the inverse of

```
(* Union the equivalence classes. *)
for each $v$ in the circular list $y, y.next, y.next.next, \ldots$ do
 $v.root := x$
end;
$x.next, y.next := y.next, x.next$;
$x.size := x.size + y.size$;
```

is

```
$x.size := x.size - y.size$;
$x.next, y.next := y.next, x.next$;
for each $v$ in the circular list $y, y.next, y.next.next$, $\ldots$, do
  $v.root := y$
end
```

The next part to invert is:

```
(* Remove old signatures. *)
for each parent $p$ of $w$'s equivalence class do
  if $p$ is a congruence root then
    Remove $((p.car.root.id, p.cdr.root.id), p)$ from the signature table
  end;
```

In addition to inverting the code segment above, the following code also inverts the effects 2b, whose inversion was postponed earlier.

```
for each parent $p$ of $w$'s equivalence class do
  var $cg := p.cgPtr$ in
    if $p = cg$ or
        $(p.car.root \neq cg.car.root$ or $p.cdr.root \neq cg.cdr.root)$
    then
      Insert $((p.car.root.id, p.cdr.root.id), p)$ into the signature table;
      $p.cgPtr := p$
    end
  end
```

```
        **end**
```

This code inserts entries into the signature table under two conditions. The first condition

$$p = cg$$

applies in the case that p was a congruence root both before and after the pertinent merge. The second condition

$$(p.car.root \neq cg.car.root \textbf{ or } p.cdr.root \neq cg.cdr.root)$$

applies in the case that p was a congruence root before, but not after, the pertinent merge. The assignment $p.cgPtr := p$ ensures that p is a congruence root again once the merge is undone.

Next, note that $x.distClasses$ and $y.distClasses$ must have been disjoint before the pertinent merge, since otherwise x and y would have been unmergeable. So the union

```
(* Merge distinction classes. *)
$x.distClasses := x.distClasses \cup y.distClasses$
```

is inverted by the set difference

$$x.distClasses := x.distClasses - y.distClasses$$

The next segment to be inverted is:

```
(* Merge forbid lists. *)
**if** $y.forbid \neq$ **nil then**
   **if** $x.forbid =$ **nil then**
      $x.forbid := y.forbid$
   **else**
      $x.forbid.link, y.forbid.link := y.forbid.link, x.forbid.link$
   **end**
**end**;
```

The inverse is:

```
**if** $x.forbid = y.forbid$ **then**
   $x.forbid :=$ **nil**
**elsif** $y.forbid \neq$ **nil then**
   $x.forbid.link, y.forbid.link := y.forbid.link, x.forbid.link$
**end**;
```

The inversion of

```
(* Maintain sets for pattern-element optimizations. *)
```

```
    Update $gpc$, $gpp$, $lbls(x)$ and $plbls(x)$ as described in Section 5.3.3.
```

is described in Section 5.3.3.

The next action of *Merge* to be undone is the propagation of equalities and the updating of *T_unknown* fields:

```
for each ordinary theory $T$ do
    (* maintain PROPAGATION TO $T$ *)
    if $x.T\_unknown \neq$ nil and $y.T\_unknown \neq$ nil then
        Propagate the equality of $x.T\_unknown$ with $y.T\_unknown$ to $T$
    end
    (* maintain ROOT $T\_unknown$ *)
    if $x.T\_unknown =$ nil then $x.T\_unknown := y.T\_unknown$ end
end;
```

There are two observations to justify the inversion of this code. First, recall from Section 4.4 that for any E-node e, if $e.T\_unknown \neq$ **nil** then $e.T\_unknown.enode$ is equivalent to e. Thus the *T_unknown* field is injective on roots, and $x.T\_unknown$ could not have been equal to $y.T\_unknown$ before the pertinent merge. Second, the ordinary theories push their own undo records, so *UndoMerge* needs no code to invert the propagation of equalities. So the inverse of the code above is:

```
for each ordinary theory $T$ do
    if $x.T\_unknown = y.T\_unknown$ then $x.T\_unknown :=$ nil end
end;
```

Finally, since x and y are local variables, *UndoMerge* needs no code to invert the following:

```
(* Make $x$ be the larger equivalence class. *)
if $x.size < y.size$ then
    $x, y := y, x$
end;
```

This completes the inversion of merge. Here is the collected complete inverse:

```
proc $UndoMerge(y : Enode) \equiv$

    var $x := y.root$ in

        $x.parentSize := x.parentSize - y.parentSize$;
        if $y.parent \neq$ nil then
            if $x.parent = y.parent$ then
                $x.parent :=$ nil
            else
                (* Unsplice the parent lists of $x$ and $y$. *)
        if $x$ and $y$ are term nodes then
          $x.parent.samecar, y.parent.samecar :=$
              $y.parent.samecar, x.parent.samecar$
        else
          $x.parent.samecdr, y.parent.samecdr :=$
              $y.parent.samecdr, x.parent.samecdr$
        end
      end
    end;

    var $w$ in
      if $x.parentSize < y.parentSize$ then
        $w := x.parent$
      else
      $w := y.parent$
      end;
      for each parent $p$ of $w$'s equivalence class do
        if $p$ is a congruence root then
          Remove $((p.car.root.id, p.cdr.root.id), p)$ from the signature table
        end;
      end;

      $x.size := x.size - y.size$;
      $x.next, y.next := y.next, x.next$;
      for each $v$ in the circular list $y, y.next, y.next.next, \ldots,$ do
        $v.root := y$
      end;

      for each parent $p$ of $w$'s equivalence class do
        var $cg := p.cgPtr$ in
         if $p = cg$ or
            $(p.car.root \neq cg.car.root$ or $p.cdr.root \neq cg.cdr.root)$ then
            Insert $((p.car.root.id, p.cdr.root.id), p)$ into the signature table;
            $p.cgPtr := p$
          end
        end
      end
    end;

    $x.distClasses := x.distClasses - y.distClasses$;

    if $x.forbid = y.forbid$ then
      $x.forbid :=$ nil
    elsif $y.forbid \neq$ nil then
      $x.forbid.link, y.forbid.link := y.forbid.link, x.forbid.link$
    end;
```

```
        **for each** ordinary theory $T$ **do**
            **if** $x.T\_unknown = y.T\_unknown$ **then** $x.T\_unknown :=$ **nil end**
        **end**

    **end**

**end**
```

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 the 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. This 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. Note that swapping of *id* fields plays no role in the example; 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) Next, 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) Later, 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) Next, 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) Next, 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) Finally, 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*. We leave the other operations to the reader.

```
proc $Cons(x, y : Enode) : Enode \equiv$
  var $res$ in
    $x := x.root$;
    $y := y.root$;
    if $sigTab(x.id, y.id) \neq$ nil then
      $res := sigTab(x.id, y.id)$
    else
      $res :=$ new$(Enode)$;
      $res.car := x$;
      $res.cdr := y$;
      (* $res$ 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 $x = \lambda(f)$ 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 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 (see Section 5.1), 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, which is represented in the E-graph by the *Cons* of a symbol node 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 code and the code that constructs matching triggers.

Second, because Simplify never performs matching during plunging (Section 4.6), there is no need to update mod-times and pairsets 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 definition. Simplify also notices most (but not all) cases in which an application of such a quasi-relation becomes unmergeable with @**true**, and in these cases it instantiates and asserts the body of the definition. Similarly to the case with quantifiers, fingerprints are used to avoid asserting or denying multiple equivalent instances of the body.

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. 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 satisfiability procedure is based on the Simplex algorithm invented by George Dantzig and widely used in Operations Research. As a consequence, our procedure shares with the Simplex algorithm a worst case behavior that would be unacceptable, but that does not seem to arise in practice.

The details of our procedure are different from the details of most Simplex algorithm implementations, because we need an incremental, resettable, equality-propagating version, and also because in our application, unknowns are not restricted to be non-negative, as they seem to be throughout the rest of the vast literature of the Simplex Algorithm. We therefore describe our version of the algorithm "from scratch" rather than merely offering a reference to some version described in the literature and leaving it to the reader to figure out the necessary changes to adapt it for our application. Our description in this section more or less follows the description in Section 12 of Greg Nelson's revised Ph.D. thesis [Nelson 1981], but our presentation is smoother and corrects an important bug in Nelson's original version. Our description is also somewhat discrepant from the actual code in Simplify, since we think it more useful to the reader to present the module as it ought to be rather than 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 atomic formulas, 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 these terms will all have the same atomic formula with the literal for the third term having opposite sense from the other two. 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 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 combinations* 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(fac : formal affine combination of connected unknowns);
(* If fac ≥ 0 is consistent with Asserted, set Asserted to fac ≥ 0 ∧
Asserted and propagate equalities as required to maintain PROPAGA-
TION FROM Simplex. Otherwise set refuted to true. *)
```

```
proc AssertEQ(fac : formal affine combination of connected unknowns);
(* If fac = 0 is consistent with Asserted, set Asserted to fac = 0 ∧
Asserted and propagate equalities as required to maintain PROPAGA-
TION 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 $UnknownForEnode(e : Enode) : Unknown$;
(* 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*. *Pop* logically restores *Asserted*, but may scramble the row and column owners.

8.2 The tableau

The *Simplex tableau* consists of

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

—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 $u.restricted$.

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 $\geq$, 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*

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

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* $u \geq 0$ for each unknown u for which $u.restricted = \mathbf{true}$. Such u's are said to be *restricted*. 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.

$y[0] = \mathit{Zero}$
$a[0,j] = 0$, 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.

$$\begin{array}{r|cccc} & 1 & {}^{\nabla}m^{\geq} & n^{\geq} & p \\ \hline \mathit{Zero} & 0 & 0 & 0 & 0 \\ w^{\geq} & 0 & 1 & -1 & 2 \\ z & -1 & -1 & -3 & 0 \end{array} \qquad (11)$$

The tableau (11) represents the tableau constraints:

$$\begin{aligned} & m \geq 0 \\ & n \geq 0 \\ & w \geq 0 \\ & \mathit{Zero} = 0 \\ & w = m - n + 2p \\ & z = -m - 3n - 1 \end{aligned}$$

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:

$\mathit{Asserted} = (\exists \mathit{unc} : \mathit{TableauSoln})$
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 $y[i]$, $y[i].\mathit{ownsRow}$ is **true** and $y[i].\mathit{index} = i$, and for any column owner $x[j]$, $x[j].\mathit{ownsRow}$ is **false** and $x[j].\mathit{index} = j$.

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

```
proc $\mathit{UnknownForEnode}(e : \mathit{Enode}) : \mathit{Unknown} \equiv$
  if $e.\mathit{Simplex\_Unknown} \neq$ nil then
    return $e.\mathit{Simplex\_Unknown}$
  end;
  $m := m + 1$;
```

```
    $x[m]$ := **new**($Unknown$);
    Push ($deallocate, x[m]$) onto the undo stack;
    $x[m].enode := e$
    $x[m].ownsRow$ := **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 combinations of unknowns as arguments, but the hearts of their implementations operate on individual unknowns. We therefore introduce a procedure that converts an affine combination into an equivalent unknown, more precisely:

```
**proc** $UnknownForFAC$($fac$ : formal affine combination of unknowns) : $Unknown$;
(* Allocates and returns a new row owner constrained to be equal to $fac$.
*)
```

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

```
**proc** $UnknownForFAC$($fac$ : formal affine combination of unknowns) : $Unknown \equiv$
    $n := n + 1$;
    $y[n]$ := **new**($Unknown$);
    Push ($deallocate, y[n]$) on the undo stack;
    $y[n].ownsRow$ := **true**;
    $y[n].index := n$;
    **let** $fac$ be the formal affine sum $k_0 + k_1 \times v_1 + \cdots + k_p \times v_p$ **in**
        $a[n, 0] := k_0$;
        **for** $j := 1$ **to** $m$ **do**
            $a[n, j] := 0$
        **end**;
        **for** $i$ :=1 **to** $p$ **do**
            **if** $v_i.ownsRow$ **then**
                **for** $j := 0$ **to** $m$ **do**
                    $a[n, j] := a[n, j] + k_i \times a[v_i.index, j]$
                **end**
            **elseif** $v_i.index > dcol$ **then**
                $a[n, v_i.index] := a[n, v_i.index] + k[v_i.index]$
            (* else $v_i$ 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 $a[i, 0]$.

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 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 v 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 is not guaranteed to increase the sample value of v, 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*(*fac*) in a nutshell:

> Create a row owner u whose row constraint is equivalent, given the existing constraints, to $u = fac$, 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 $fac \geq 0$, so restrict u. In case (2), the inequality $fac \geq 0$ 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 $a[u.index, 0] \le 0$ do
    $(j, k)$ := FindPivot(u.index);
    if $j = u.index$ then
      Pivot(j, k)
      (* u is manifestly unbounded. *)
      return 1
    elseif $(j, k) = (-1, -1)$ 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 change to the sample value of $y[i]$ caused by the pivot will be $a[i, j]$ times the change in the sample value of $x[i]$ (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 $x[i]$ 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 $FindPivot(i : \textbf{integer}) : (\textbf{integer} \times \textbf{integer}) \equiv$
  var $j$, $sgn$ in
    if there exists a $k > dcol$ such that $\neg\, x[k].restricted$ and $a[i, k] \neq 0$ then
      Choose such a $k$;
      $(j, sgn) := (k, a[i, k])$
    elseif 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 $champ := i$, $score := \infty$ in
      for each row $h$ such that $h \neq i$ and $y[h].restricted$ do
        if $sgn \times a[h, j] < 0 \ \wedge\ |a[h, 0]/a[h, j]| < score$ 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 $ToRow(u : Unknown) \equiv$
(* 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 $ToRow(u : Unknown) \equiv$
  if $u.ownsRow$ then
    return
  end;
  var $j = u.index$, $champ$, $score := \infty$ in
    for each row $h$ such that $y[h].restricted \land a[h, j] < 0$ 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 chosen row owner as a column owner with sample value 0.

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

```
proc $AssertGE(fac$ : formal affine combination of connected unknowns$) \equiv$
  var $s := UnknownForFAC(fac)$ in
    var $sgmx := SgnOfMax(s)$ in
      if $sgmx < 0$ then
        $refuted :=$ true;
        return
      else
```

```
            $u.restricted$ := true;
            Push ($unrestrict, u$) onto the undo stack;
            if $sgmx = 0$ then
               $CloseRow$(s.$index$);
            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 max-
imized 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.

By the assumption that $f = 0$ holds over the intersection of *TableauSoln* with the half-space $u \geq 0$, it follows that $f(R) = 0$ (since $u(R) > 0$) and $f(Q) = 0$ (since $u(Q) = 0$). Since the affine function f is zero at two distinct points, Q and R, on PR, it must be zero on the entire line, including at P. Q.E.D.

It would be possible to implement *AssertEQ* 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 non-zero 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 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 non-zero 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 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 combination 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 combinations are identical in all coefficients. It is straightforward to check that coefficients are identical precisely when the unknowns are manifestly equal. Q.E.D.

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,

$$f = 0 \text{ over } \mathit{TableauSoln} \;\Leftrightarrow\; f = 0 \text{ over } \mathit{HPlaneSoln} \ .$$

Proof. The "$\Leftarrow$" part is a consequence of the fact that *TableauSoln* $\subseteq$ *HPlaneSoln*.

The " $\Rightarrow$ " 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. Q.E.D.

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 $KillCol(j : \textbf{integer})$
(* 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 $Propagated$. Undoably adds the dead column
constraint $u = 0$, 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 $KillCol(j : \textbf{integer}) \equiv$
  Push $reviveCol$ onto the undo stack;
  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 $a[i, dcol] \neq 0$ 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 $CloseRow(i) \equiv$
  for each $j > dcol$ such that $a[i, j]$ is non-zero (* equivalently, is negative *) do
    $KillCol(j)$
  end;
  var $R :=$ the set of all restricted unknowns with sample value 0 but not manifestly equal to $Zero$ in
    while $R \neq \{\}$ 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 $a[u.index, j]$ is non-zero (* equivalently, is negative *) **do**
                $KillCol(j)$
              **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 non-minimal 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 subsection than in the rest of our description of the Simplex module.

As remarked above, $AssertEQ$ 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 $(v > 0 \ \wedge \ u = 0)$ 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 $v(R) > 0$. Q.E.D.

We will provide an improved implementation of *AssertEQ* 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 *AssertEQ* as follows:

```
proc $AssertEQ(fac$ : formal affine combination of connected unknowns$) \equiv$
  var $u := UnknownForFAC(fac)$ in
    if $u$ is manifestly equal to $Zero$ then
      return
    end;
    var $sgmx := SgnOfMax(u)$ in
      if $sgmx < 0$ then
        $refuted :=$ true;
        return
      elseif $sgmx = 0$ then
        $CloseRow(u.index)$;
        return
      else
        (* Sign of max of $fac$ 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 $-fac$. *)
        $sgmx := SgnOfMax(u)$ in
        if $sgmx < 0$ then
          $refuted :=$ true;
```

```
                **return**
            **elseif** $sgmx = 0$ **then**
                $CloseRow(u.index)$;
                **return**
            **else**
                (* Sign of max of $-fac$ 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 *AssertEQ* 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 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 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 subsection, 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 $(\mathit{unrestrict}, u)$:

```
$u.\mathit{restricted} := \mathit{false}$
```

To process an undo record of the form *reviveCol*:

```
$\mathit{dcol} := \mathit{dcol} - 1$
```

To process an undo record of the form $(\mathit{deallocate}, u)$:

```
if $u.\mathit{ownsRow}$ then
    copy row $n$ to row $u.\mathit{index}$;
    $y[u.\mathit{index}] := y[n]$;
    $n := n - 1$
elseif $u$'s column is identically 0 then
    copy column $m$ to column $u.\mathit{index}$;
    $x[u.\mathit{index}] := x[m]$;
    $m := m - 1$
else
    perform some pivot in $u$'s column that preserves feasibility;
    copy row $n$ to row $u.\mathit{index}$;
    $y[u.\mathit{index}] := y[n]$;
    $n := n - 1$
end;
if $u.\mathit{enode} \neq$ nil then
    $u.\mathit{enode}.\mathit{Simplex\_Unknown} :=$ nil;
    $u.\mathit{enode} :=$ nil
end
```

In arguing the correctness of this undo action, we observe that there are two case in which an unknown is allocated: *UnknownForEnode* allocates a completely unconstrained unknown and connects it to an E-node, and *UnknownForFAC* 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 *UnknownForFAC*, 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 $0 \leq b \wedge b \leq 1$, after which $\neg b$ is encoded $1-b$, and $b \vee c \vee d$ is encoded $b + c + d > 0$. Thus 3SAT is encoded. QED.

For integer linear inequalities that arise from such an encoding, it seems extremely likely (as likely as $P \neq NP$) 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 $\neg\, a \leq b$, 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:

$$i \leq n \ \wedge\ f(i) \neq f(n) \Rightarrow i \leq n - 1$$

The reason this formula is proved is that the only possible counterexample context is

$$i \leq n, \quad f(i) \neq f(n), \quad \neg\, i \leq n - 1$$

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, the prover will try to refute the possibility $i = L$ (not try so hard as to do a case split, but Simplify will do non-unit matching in the attempted refutation), and if it is successful, it will then strengthen the upper 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 context. 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 was on a goal 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*.

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 paper.

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.

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 check the next routine. The default time limit used by ESC/Java is 300 seconds.

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 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 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. These tests are available on the web [Detlefs et al. 2003a].

Sections 9.1 through 9.12 show the effect of setting switches or otherwise disabling various optimizations that Simplify uses. Section 9.13 through 9.15 discuss more general performance issues. In the first twelve subsections, we don't, of course, test every combination of optimizations, but we have tried to test each optimization against the *baseline* behavior in which all optimizations are enabled. Figure 3 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 time during the proof that the matching depth was incremented, and the maximum matching depth reached.

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.

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. 3. Baseline performance data for the small test suite.

9.1 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 test, but also more expensive. By default Simplify plunges on each literal of each clause produced by matching a non-unit rule: untenable literals are deleted from the clause, and if any literal's negation is found untenable, the entire clause is deleted.

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 4 compares, on the small test suite, the performance of Simplify with plunging disabled to the baseline. The figure has the same format as Figure 3 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.

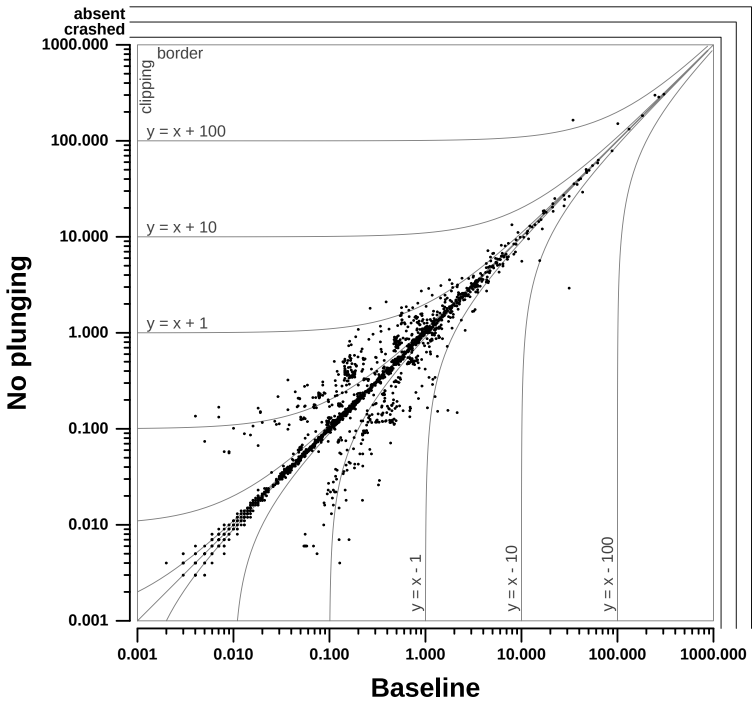
Figure 5 below illustrates the performance effect of plunging on each of the routines in the ESC/Java front end.

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 VC 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

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. 4. Performance data for the small test suite with plunging disabled.



Checked 2331 routines in 4030 sec. (vs. 3841 sec. for baseline)
Includes 1 timeout.

Fig. 5. Effect of disabling plunging on the routines in the ESC/Java front end.

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. 6. Performance data for the small test suite with the mod-time optimization disabled.

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 non-plunging version of) *Refine*.

2 Matching on unit rules.

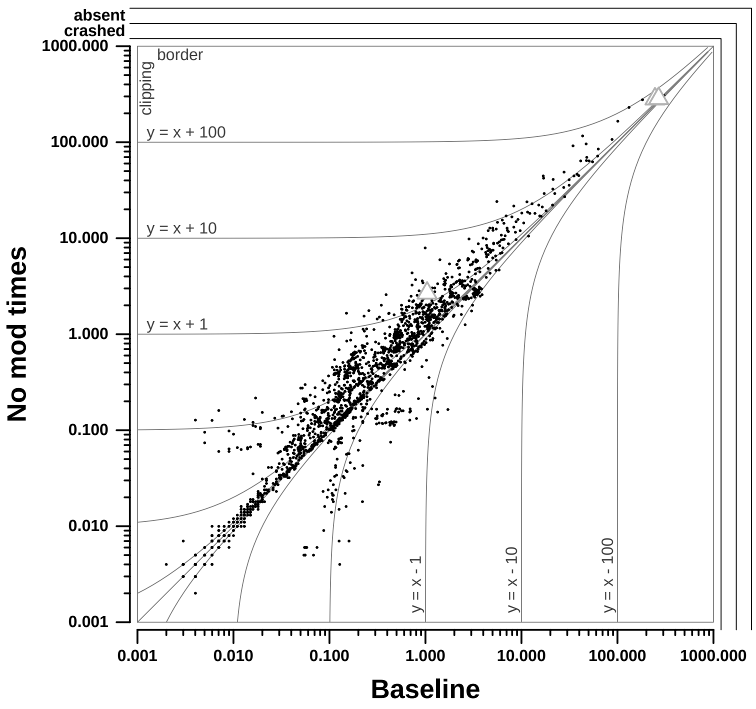
3 Match on non-unit rules (but do not plunge on their instances).

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.

9.2 The mod-time optimization

As described in Section 5.3.2, 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 6 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



Checked 2331 routines in 5377 sec. (vs. 3841 sec. for baseline)
Includes 3 timeouts.

Fig. 7. Effect of disabling the mod-time optimization on the routines in the ESC/Java front end.

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 7 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 7, the two triangles in the upper right corner are cases where disabling the mod-time optimization caused timeouts. The hard-to-see 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 off 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.

9.3 The pattern-element optimization

As described in Section 5.3.3, 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 8 and 9, the results on the examples from program checking are similar to those for the mod-time optimization–either improvements (sometimes

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. 8. Performance data for the small test suite with the pattern-element optimization disabled.

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 10 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.

9.4 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 of the literal to **false**, and also, if it is a literal of the E-graph (except for non-binary distinctions) or of an ordinary theory, denying it by a call to the appropriate assert method. This behavior can be changed by setting PROVER_SUBSUMPTION to "None", causing Simplify to set the status of the literal, but not to assert its negation. Figure 11 shows that the performance data for this switch are inconclusive. The data on the front end test suite (not shown) are also inconclusive.

9.5 Merit promotion

As described in Section 5.1 Simplify uses merit promotion: When a *Pop* reduces the matching depth, Simplify promotes the highest scoring clause from the deeper matching level to the shallower level.

Figures 12 and 13 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

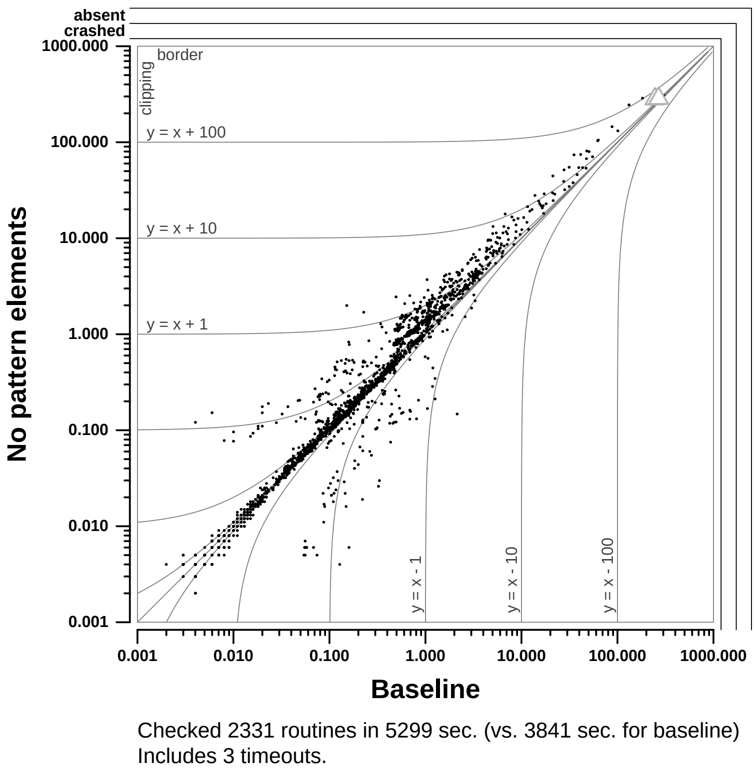


Fig. 9. Effect of disabling the pattern-element optimization on the routines in the ESC/Java front end.

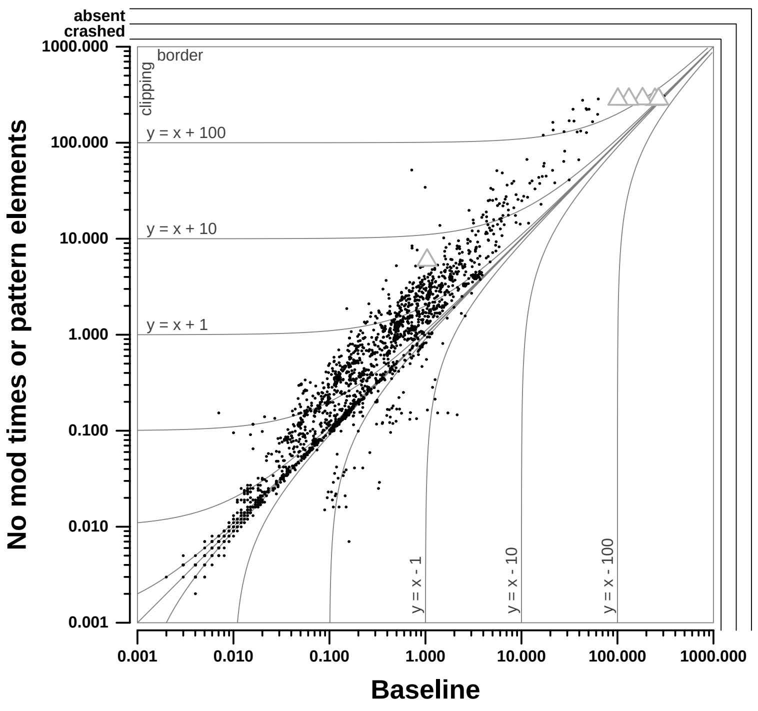
that counts on only one (which in fact is `getNextPragma`). The total time increases by about ten percent.

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.

9.6 Immediate promotion

As described in Section 5.1 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 non-unit select-of-store axiom.

Figure 14 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 15 which is where the expensive VC's are, outliers above the diagonal clearly outnumber outliers below the diagonal. Furthermore, the heuristic reduces the total time on the front end by about five percent.



Checked 2331 routines in 9544 sec. (vs. 3841 sec. for baseline)
Includes 6 timeouts.

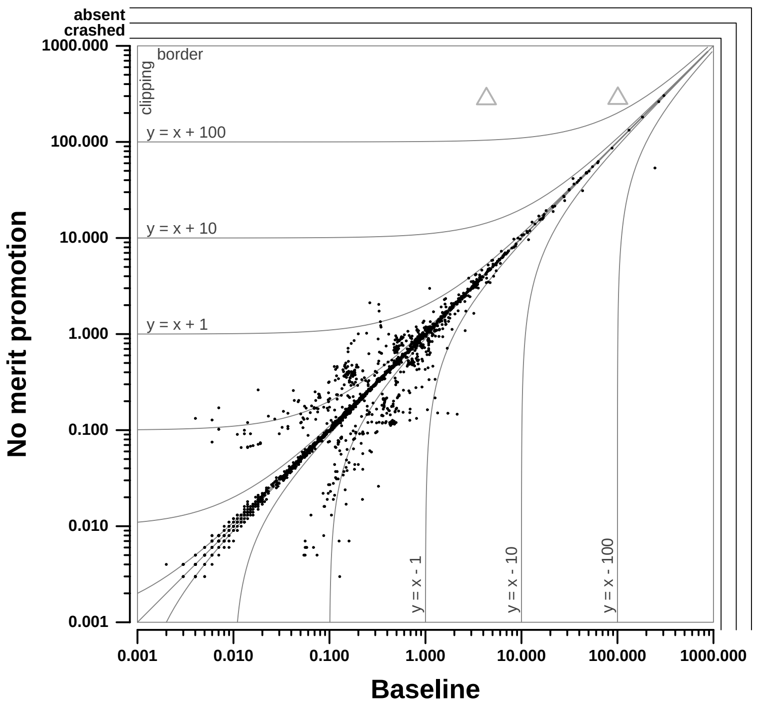
Fig. 10. Effect of disabling both the mod-time and pattern-element optimizations on the routines in the ESC/Java front end.

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. 11. 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. 12. Performance data for the small test suite with merit promotion disabled.

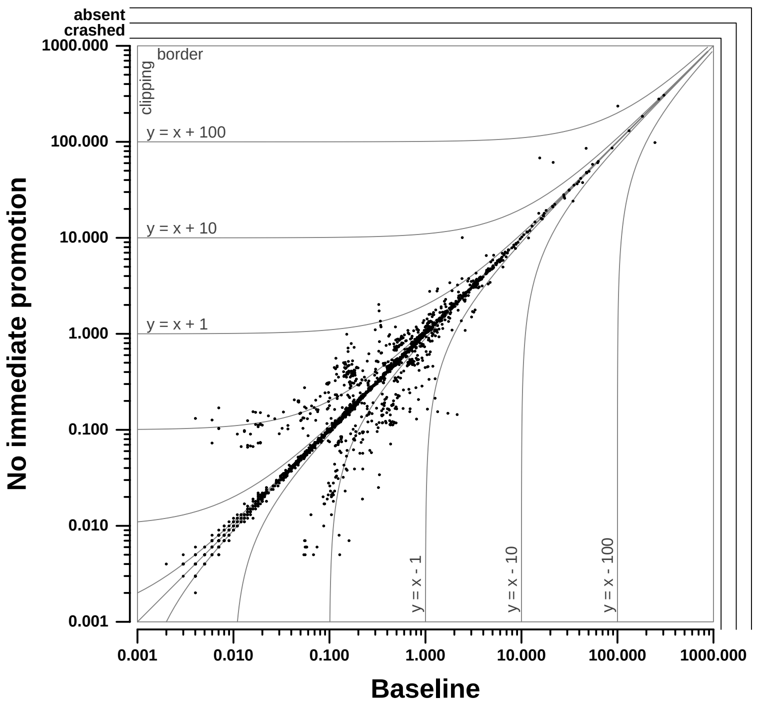


Checked 2331 routines in 4121 sec. (vs. 3841 sec. for baseline)
Includes 3 timeouts.

Fig. 13. Effect of disabling merit promotion on the routines in the ESC/Java front end.

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. 14. 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. 15. Effect of disabling immediate promotion on the routines in the ESC/Java front end.

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. 16. Performance data for the small test suite with the activation heuristic disabled (so that triggers are matched against all E-nodes, regardless of whether those E-nodes represent terms in asserted basic literals).

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.

9.7 Activation

As described in Section 5.1, Simplify uses the activation heuristic, which sets the *active* bit in those E-nodes that represent subterms of atomic formulas that are currently asserted or denied, and restricts the matcher to find only those instances of matching rule triggers that lie in the active portion of the E-graph.

Figure 16 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), several other tests slow down significantly, and only one (`checkTypeDeclOfSig`) 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.

9.8 Scoring

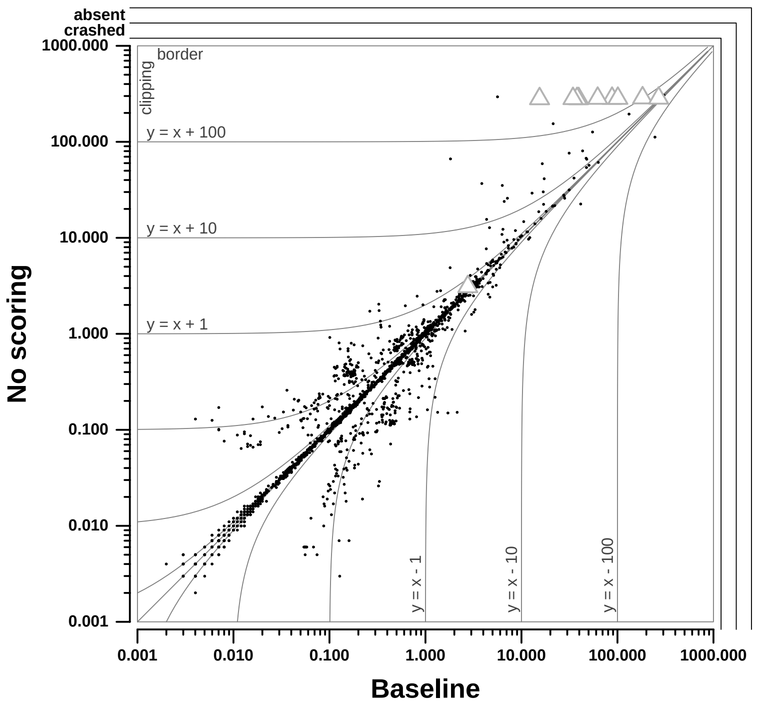
As described in Section 3.6, Simplify uses a scoring heuristic, which favors case splits that have recently produced contradictions. Figures 17 and 18 show that scoring is effective.

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, the scoring had mixed effects on cheap problems, but was very helpful for expensive problems. Turning it off caused eight problems to time out and increased the total time by sixty percent.

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. 17. 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. 18. Effect of disabling scoring on the routines in the ESC/Java front end.

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. 19. Performance data for the small test suite with the E-graph status test disabled.

9.9 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 bits only, when refining clauses. Figures 19 and 20 show that the E-graph test is effective, especially on expensive problems.

9.10 Distinction classes

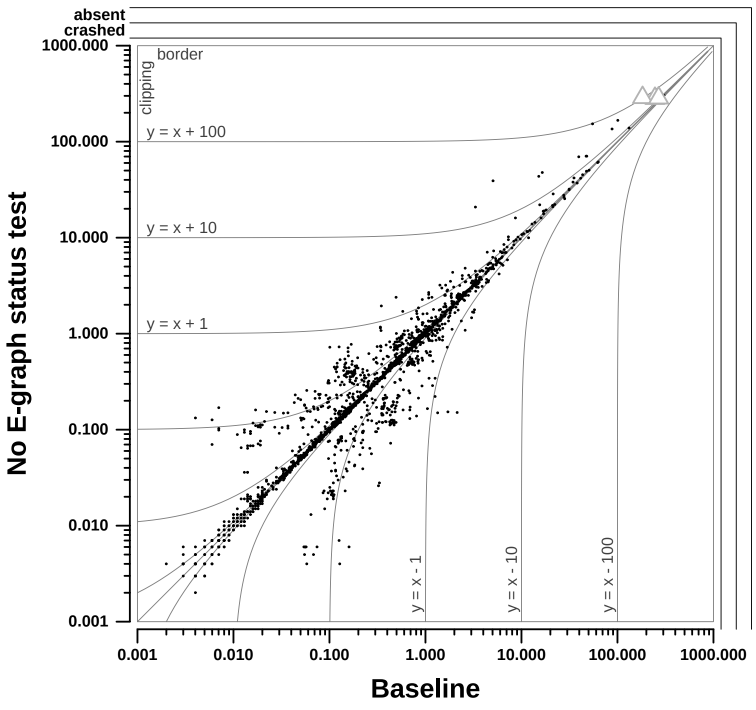
As described in Section 7, Simplify uses distinction classes, a space-efficient technique for asserting the pairwise distinctness of a set of more than two items. Interestingly enough, Figures 21 and 22 show the performance effects of distinction classes are negligible. We do not know what causes the strange asymmetrical pattern of dots in the front end test.

9.11 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 VC's essentially as fast as it proves valid unlabeled VC's. This experiment was performed on the small test suite by setting a switch that causes Simplify to discard labels at an early phase. Figure 23 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 24 shows the results: in all but one of the examples where the difference was significant, labels were faster than error variables.

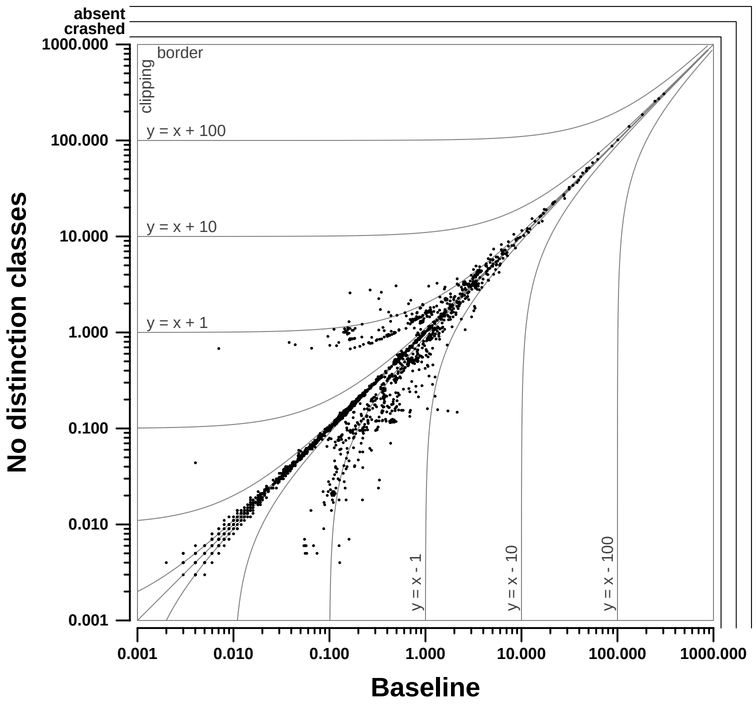


Checked 2331 routines in 4560 sec. (vs. 3841 sec. for baseline)
Includes 4 timeouts.

Fig. 20. Effect of disabling the E-graph status test on the routines in the ESC/Java front end.

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. 21. 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).



Checked 2331 routines in 3964 sec. (vs. 3841 sec. for baseline)
Includes 1 timeout.

Fig. 22. Effect of disabling distinction classes on the routines in the ESC/Java front end.

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. 23. 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. 24. Performance data for the small test suite with labels replaced by error variables.

9.12 Simplex redundancy filtering

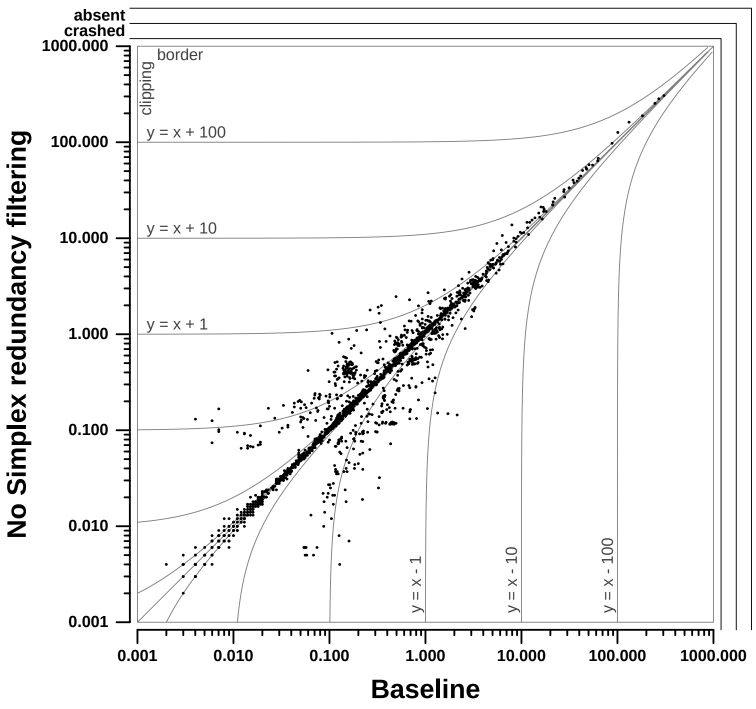
As described in Section 8.3, Simplify uses Simplex redundancy filtering to avoid processing certain trivially redundant arithmetic assertions. Figure 25 shows the effect of disabling this optimization on the front-end test suite. The aggregate effect of the optimization is an improvement of about 10 percent.

9.13 Fingerprinting matches

As described in Section 5.1, 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. The mod-time and pattern-element 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.

9.14 Performance of the Simplex module

To represent the Simplex tableau, 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 (for example the one described by Knuth [Knuth 1968, Section 2.2.6]) might improve performance, but more careful measurements would be required to tell for sure.



Checked 2331 routines in 4120 sec. (vs. 3841 sec. for baseline)
Includes 1 timeout.

Fig. 25. Effect of disabling simplex redundancy filtering on the routines in the ESC/Java front end.

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 26 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: there is a row for each of several tableau size ranges (where size is defined as $\max(m, n)$) and a column for each of several pivot count ranges; each entry in the figure is the number of calls to *SgnOfMax* for which the tableau size was in its row's size range and the number of pivots was in its column's pivot count range.

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, the greatest number of pivots performed in any call to *SgnOfMax* was 686.

9.15 Performance of the E-graph module

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

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%)	(.0003%)

Fig. 26. Distribution of the 264630 calls to *SgnOfMax* in the baseline run of the front end test-suite, according to size of the tableau and number of pivots required.

unweighted average rate is 6100 merges/second.

Roughly speaking, the E-graph typically contains a few thousand active nodes. Over the small test suite, the maximum number of active nodes over the course of the proof ranged from 675 to 13000, and averaged 3400.

We conclude this subsection with three further observations about the performance of the E-graph module, as implemented according to Section 7. Due to time pressure, we have not checked these observations as carefully as we would like, but we found them to be true of three representative examples in the small test suite.

The central idea of the Downey-Sethi-Tarjan congruence-closure algorithm is to find new congruences by rehashing the shorter of two parent lists. This achieves an asymptotic worst case time of $O(n \log n)$. The idea is very effective in practice. The total number of nodes rehashed is often smaller than the number of merges.

There are more signature table operations due to *Cons* than to rehashing in *Merge* and *UndoMerge*.

A majority of calls to *Cons* find and return an existing node.

9.16 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

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 [Shostack 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 [Shankar 2003].

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 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.

We and others have exploited these observations with a new architecture that we have called "explicated clauses" [Flanagan et al. 2003], Dill's group has called "conflict clauses" [Barrett et al. 2002a], and the SRI group has called "lemmas on demand" [de Moura and Ruess 2002]. The new architecture uses the domain-specific decision procedures to repeatedly check the solution obtained by a fast propositional SAT solver against the semantics of the built-in theories, manifesting any conflict as a clause that is added to the SAT solver's constraints, until eventually either obtaining an unsatisfiable SAT problem or finding a solution that is consistent with both the propositional constraints and the built-in theories.

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 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 our background axioms 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.

sectionConclusions

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 [Schmitt 2003; 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: correctly undoing merge operations in the E-graph and 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.

REFERENCES

Agesen, O., Detlefs, D. L., Flood, C. H., Garthwaite, A. T., Martin, P. A., Shavit, N. N., and Jr., G. L. S. 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. Feb.

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*. 203–213. Snowbird, Utah.

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 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.

Crocker, S. 1988. Comparison of Shostak's and Oppen's solvers. Unpublished manuscript.

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.*

Detlefs, D. and Moir, M. 2000. Mechanical proofs of correctness for dcas-based concurrent deques. Available via `http://research.sun.com/research/jtech/pubs/00-deque1-proof.html`.

Detlefs, D., Nelson, G., and Saxe, J. B. 2003a. Simplify benchmarks. Available via `http://www.hpl.hp.com/research/src/esc/simplify_benchmarks.tar.gz`.

Detlefs, D., Nelson, G., and Saxe, J. B. 2003b. Simplify source code. Available via `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. Dec.

Downey, P. J., Sethi, R., and Tarjan, R. E. 1980. Variations on the common subexpression problem. *JACM 27,* 4 (October), 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*. To appear.

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)*. 234–245. Berlin.

Galler, B. A. and Fischer, M. J. 1964. An improved equivalence algorithm. *CACM 7,* 5, 301–303.

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)*. 526–530. Munich.

Joshi, R., Nelson, G., and Randall, K. 2002. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM 2000 Conference on Programming Language Design and Implementation*. 304–314. Berlin.

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.

Marcus, L. 1981. A comparison of two simplifiers. Microver Note 94, SRI. Jan.

Millstein, T. 1999. Toward more informative ESC/Java warning messages. In Compaq SRC Technical Note 1999-003. Available on the web at http://gatekeeper.research.compaq.com/pub/DEC/SRC/technical-notes/SRC-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. *Journal of the ACM 27,* 2 (Apr.), 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.

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.

Schmitt, P. H. 2003. Personal communication (email message to Greg Nelson).

Shankar, N. 2003. Personal communication (email message to James B. Saxe).

Shostack, 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.

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, 103–120. Munich.

Zhang, H. 1997. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*. 272–275.