Simplification by Cooperating Decision Procedures

GREG NELSON and DEREK C. OPPEN
Stanford University

A method for combining decision procedures for several theories into a single decision procedure for their combination is described, and a simplifier based on this method is discussed. The simplifier finds a normal form for any expression formed from individual variables, the usual Boolean connectives, the equality predicate =, the conditional function **if-then-else**, the integers, the arithmetic functions and predicates +, −, and ≤, the Lisp functions and predicates **car**, **cdr**, **cons**, and **atom**, the functions **store** and **select** for storing into and selecting from arrays, and uninterpreted function symbols. If the expression is a theorem it is simplified to the constant **true**, so the simplifier can be used as a decision procedure for the quantifier-free theory containing these functions and predicates. The simplifier is currently used in the Stanford Pascal Verifier.

Key Words and Phrases: program verification, program manipulation, theorem proving, decidability, simplification
CR Categories: 4.12, 5.21, 5.24, 5.25, 5.7

1. INTRODUCTION

Program verifiers, symbolic evaluators, program transformation systems, and similar high-level "program manipulation" systems will make the programming process more automatic, less error prone, and cheaper—if they can ever be made practical. The main reason they have not yet progressed beyond the experimental stage is that they depend on mechanical theorem provers, and these have been too slow and unreliable.

Theorem provers are required by program manipulation systems to verify routine facts about numbers, arrays, lists, and other common data structures. We give two examples of the sort of theorems which arise. First, consider the loop

$$\textbf{for } i \leftarrow 0 \textbf{ to } n \textbf{ do } v[i] \leftarrow f(i),$$

where f is a function with no side effects. Usually the test $i = 0$ compiles better than the test $i = n$, so a program transformation system might try to change the loop to run i from n to 0 instead of from 0 to n. To justify this transformation, it

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, by the National Science Foundation under Contract MCS78-02835, and by the Fannie and John Hertz Foundation.
Authors' address: Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, CA 94305.

must prove that two consecutive loop executions "commute"; that is, that they can be executed in either order with the same effect. Let **store**(v, i, e) denote the array whose ith element is e and whose jth element, for $j \neq i$, is $v[j]$. The theorem which must be proved is

$$\textbf{store}(\textbf{store}(v, i, f(i)), i + 1, f(i + 1)) = \textbf{store}(\textbf{store}(v, i + 1, f(i + 1)), i, f(i)).$$

The second example is from program verification. If a random integer in the range $[a, b]$ is generated from a random real number λ in the range $[0, 1)$ by the formula $\lfloor a + \lambda(b - a + 1) \rfloor$, the verification that the output is in the correct range may require proving the "theorem":

$$0 \leq \lambda \wedge \lambda < 1 \supset a \leq \lfloor a + \lambda(b - a + 1) \rfloor \wedge \lfloor a + \lambda(b - a + 1) \rfloor \leq b.$$

A verifier will reject this theorem, alerting the user to the fact that critical assumptions are missing—the assumptions that a and b are integers and that $a \leq b$.

Neither of the above theorems is mathematically interesting. Mechanical reasoning is used in program manipulation to verify routine facts or to catch slippery errors, not to prove mathematically interesting theorems.

Notice that the formulas above are "quantifier free"—their variables are implicitly universally quantified. The theorem-proving methods described in this paper apply only to quantifier-free formulas.

An algorithm for determining whether a formula is valid in a logical theory is called a *decision procedure* for the theory. Decision procedures are known for many quantifier-free theories important in program manipulation. For example, there are decision procedures for the theory of integers under + and $\leq$, for the theory of arrays under **store** and **select**, and for the theory of equality with uninterpreted function symbols. Unfortunately, the expressions which arise in program manipulation often do not fall within any of these naturally defined theories—they usually involve "mixed" terms containing functions and predicates from several theories. For example, the first theorem above contained the arithmetic function +, the array function **store**, and the uninterpreted function symbol f.

There has been some research on decision procedures for specific quantifier-free theories with mixed terms: Kaplan [4] gives a decision procedure for the theory of arrays with constant indices, Shostak [11] for Presburger arithmetic with uninterpreted function symbols, and Suzuki and Jefferson [12] for Presburger arithmetic with arrays and uninterpreted function symbols.

In this paper we give a general method for combining decision procedures for two quantifier-free theories into a single decision procedure for their combination, which contains the functions and predicates of both theories. The method is based on a technique which we call *equality propagation*. Using this technique, we have implemented a simplifier which finds a simplified, normal form for any expression formed from individual variables, the usual Boolean connectives, the equality predicate =, the conditional function **if-then-else**, the integers, the arithmetic functions and predicates +, −, and $\leq$, the Lisp functions and predicates **car**, **cdr**, **cons**, and **atom**, the functions **store** and **select**, and uninterpreted function symbols. If the expression is a theorem, it is simplified to **true**; if it is unsatisfiable, it is simplified to **false**. If it is neither, or if it is not a Boolean-

$$
\begin{array}{l}
(\neg(x \le y) \wedge \\
(x - 1 \le y \wedge tak(x - 1, y, z) = y \vee \\
y < x - 1 \wedge \\
\quad (y \le z \wedge tak(x - 1, y, z) = z \vee \\
\quad z < y \wedge tak(x - 1, y, z) = x - 1)) \wedge \\
(y - 1 \le z \wedge tak(y - 1, z, x) = z \vee \\
z < y - 1 \wedge \\
\quad (z \le x \wedge tak(y - 1, z, x) = x \vee \\
\quad x < z \wedge tak(y - 1, z, x) = y - 1)) \wedge \\
(z - 1 \le x \wedge tak(z - 1, x, y) = x \vee \\
x < z - 1 \wedge \\
\quad (x \le y \wedge tak(z - 1, x, y) = y \vee \\
\quad y < x \wedge tak(z - 1, x, y) = z - 1)) \wedge \\
(tak(x - 1, y, z) \le tak(y - 1, z, x) \wedge \\
\quad tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y)) = tak(y - 1, z, x) \vee \\
tak(y - 1, z, x) < tak(x - 1, y, z) \wedge \\
(tak(y - 1, z, x) \le tak(z - 1, x, y) \wedge \\
tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y)) = tak(z - 1, x, y) \vee \\
tak(z - 1, x, y) < tak(y - 1, z, x) \wedge \\
tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y)) = tak(x - 1, y, z))) \\
\supset \\
x \le y \wedge tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y)) = y \vee \\
y < x \wedge \\
(y \le z \wedge tak(tak(x - 1, y, z)\ tak(y - 1, z, x), tak(z - 1, x, y)) = z \vee \\
z < y \wedge tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y)) = x))
\end{array}
$$

Fig. 1.

valued expression, the simplifier returns a simplified, normal form of the expression. Thus we call our program a simplifier rather than a theorem prover. The simplifier is presently used in the Stanford Pascal Verifier, an interactive system for reasoning about Pascal programs.

Here are examples of some relatively trivial simplifications:

$2 + 3*5$
17

$P \supset \neg P$
$\neg P$

$x \neq x$
false

$x = f(x) \supset f(f(f(x))) = x$
true

$\mathbf{cons}(x, y) = z \supset \mathbf{car}(z) + \mathbf{cdr}(z) - x - y = 0$
true

$x \le y \wedge y + d \le x \wedge 3*d \ge 2*d \supset V[2*x - y] = V[x + d]$
true

(In the last example, the first two literals imply that $d \le 0$, the third that $d \ge 0$. Hence $d = 0$ and $x = y$.)

The following is a somewhat artificial example involving a larger formula. Ikuo Takeuchi [13] defined the function:

$tak(x, y, z) =$ **if** $x \le y$ **then** y **else** $tak(tak(x - 1, y, z), tak(y - 1, z, x), tak(z - 1, x, y))$.

John McCarthy proved that the function satisfies

$$tak(x, y, z) = \textbf{if } x \leq y \textbf{ then } y \textbf{ else if } y \leq z \textbf{ then } z \textbf{ else } x. \quad (1)$$

A key step in the proof is the verification that if $x > y$ and (1) holds for each of the four recursive calls in *tak*, then (1) holds for $tak(x, y, z)$. We wrote *tak* as a recursive Pascal program and used the Stanford Pascal Verifier to produce the actual formula to be proved; it is shown in Figure 1. The simplifier described in this paper simplifies this formula to **true**, requiring 0.8 CPU seconds on a DEC KL-10 running compiled MACLISP.

2. THE THEORIES $\mathscr{R}$, $\mathscr{A}$, $\mathscr{L}$, AND $\mathscr{E}$

Our simplifier contains decision procedures for four quantifier-free theories: the theory of real numbers under + and ≤, the theory of arrays under **store** and **select**, the theory of list structure with **car**, **cdr**, **cons**, and **atom**, and the theory of equality with uninterpreted function symbols. We call these theories $\mathscr{R}$, $\mathscr{A}$, $\mathscr{L}$, and $\mathscr{E}$, respectively.

The theories are formalized in classical first-order logic with equality, extended to include the three-argument conditional function **if-then-else**. The symbols $=$, $\wedge$, $\vee$, $\neg$, $\supset$, **if-then-else**, $\forall$, and $\exists$ are common to all theories; we call them the *logical symbols*. Each theory is characterized in the usual way by its set of *nonlogical symbols* and *nonlogical axioms*.

The nonlogical symbols of $\mathscr{R}$ are +, −, ≤, 0, and 1. Its axioms are

$\forall x$	$x + 0 = x$
$\forall x$	$x + -x = 0$
$\forall x \forall y \forall z$	$(x + y) + z = x + (y + z)$
$\forall x \forall y$	$x + y = y + x$
$\forall x$	$x \leq x$
$\forall x \forall y$	$x \leq y \vee y \leq x$
$\forall x \forall y$	$x \leq y \wedge y \leq x \supset x = y$
$\forall x \forall y \forall z$	$x \leq y \wedge y \leq z \supset x \leq z$
$\forall x \forall y \forall z$	$x \leq y \supset x + z \leq y + z$
	$0 \neq 1$
	$0 \leq 1$

Of course, the numerals and the common relations are allowed but are formally regarded as abbreviations: 2 abbreviates 1 + 1 and $x < y$ abbreviates $\neg y \leq x$. Multiplication by integer constants is also allowed; $2 * x$ abbreviates $x + x$.

Statements which are valid over the integers but not over the reals, such as $x + x \neq 5$, are not consequences of the above axioms and will not be simplified to **true** by our simplifier. Luckily such formulas occur relatively infrequently in program manipulation. Decision procedures exist for the theory of the integers under + and ≤, but we have not tried implementing any of them, since they are considerably more complicated than decision procedures for $\mathscr{R}$. Our simplifier includes some simple heuristics for dealing with integers (such as converting $x < y$ into $x \leq y - 1$ when both variables are known to be integers) which work well in practice.

A more annoying problem is that only multiplication by constants is handled. To some extent this incompleteness is inherent, since no decision procedure can exist for the quantifier-free theory of integers under addition and multiplication (see Matiyasevich [5]). The theory of the real numbers under addition and

multiplication is decidable (see Tarski [14]), but we have not tried to implement a decision procedure for this theory.

The theory of arrays, $\mathcal{A}$, has the nonlogical symbols **store** and **select** and the axioms

$$\begin{array}{ll}
\forall v \forall e \forall i \forall j & \mathbf{select}(\mathbf{store}(v, i, e), j) = \mathbf{if}\ i = j\ \mathbf{then}\ e\ \mathbf{else}\ \mathbf{select}(v, j) \\
\forall v \forall i & \mathbf{store}(v, i, \mathbf{select}(v, i)) = v \\
\forall v \forall i \forall e \forall f & \mathbf{store}(\mathbf{store}(v, i, e), i, f) = \mathbf{store}(v, i, f) \\
\forall v \forall i \forall j \forall e \forall f & i \neq j \supset \mathbf{store}(\mathbf{store}(v, i, e), j, f) = \mathbf{store}(\mathbf{store}(v, j, f), i, e)
\end{array}$$

We write $v[i]$ for $\mathbf{select}(v, i)$. A two-dimensional array can be treated as an array of arrays, so $A[i, j]$ is shorthand for $A[i][j]$. The last three axioms are only needed if equalities between array terms are allowed.

The theory of list structure, $\mathcal{L}$, has the nonlogical symbols **car**, **cdr**, **cons**, and **atom** and the axioms

$$\begin{array}{ll}
\forall x \forall y & \mathbf{car}(\mathbf{cons}(x, y)) = x \\
\forall x \forall y & \mathbf{cdr}(\mathbf{cons}(x, y)) = y \\
\forall x & \neg\mathbf{atom}(x) \supset \mathbf{cons}(\mathbf{car}(x), \mathbf{cdr}(x)) = x \\
\forall x \forall y & \neg\mathbf{atom}(\mathbf{cons}(x, y))
\end{array}$$

That is, $\mathbf{cons}(x, y)$ is the ordered pair (x, y); $\mathbf{car}(z)$ and $\mathbf{cdr}(z)$ are the components of the ordered pair z; and $\mathbf{atom}(z)$ is true if and only if z is not an ordered pair. Notice that acyclicity is not assumed; for instance, $\mathbf{car}(x) = x$ is regarded as satisfiable.

Finally, we define the theory $\mathcal{E}$ whose nonlogical symbols are all uninterpreted function, constant, and predicate symbols. $\mathcal{E}$ has no axioms, so it is just the theory of equality.

3. TWO EXAMPLES OF EQUALITY PROPAGATION

If $\mathcal{S}$ is a theory, the terms, literals, and formulas of the language of $\mathcal{S}$ will be called $\mathcal{S}$-terms, $\mathcal{S}$-literals, and $\mathcal{S}$-formulas. (A *literal* is an atomic formula or its negation.) For example, $x = y$ and $x \leq y + 3$ are $\mathcal{R}$-literals but $x \leq \mathbf{car}(y)$ is not.

A *satisfiability procedure* for a theory $\mathcal{S}$ is a decision procedure for determining the satisfiability of conjunctions of $\mathcal{S}$-literals. (The general quantifier-free decision problem for $\mathcal{S}$ can easily be reduced to this problem.) Our simplifier contains satisfiability procedures for the theories described Section 2. We use $\mathcal{R}$, $\mathcal{A}$, $\mathcal{L}$, and $\mathcal{E}$ for the names of the satisfiability procedures as well as for the names of the theories.

A formula F *entails* a formula G if G is a logical consequence of F; that is, if $F \supset G$ is a theorem of first-order logic. A formula F entails a formula G *within a theory* $\mathcal{S}$ if $F \supset G$ is a theorem in $\mathcal{S}$. If the context is clear, we omit specifying the theory; for instance, we say that $x - y = 0$ entails $x = y$ without specifying "in the theory $\mathcal{R}$."

The satisfiability procedures in our simplifier also detect, and "propagate" to the other satisfiability procedures, certain equalities entailed by the conjunction being decided. The detection of these equalities is the key to our method of combining decision procedures. To illustrate this process, we describe how $\mathcal{R}$, $\mathcal{L}$, and $\mathcal{E}$ together detect the unsatisfiability of the following conjunction F:

$$x \leq y \wedge y \leq x + \mathbf{car}(\mathbf{cons}(0, x)) \wedge P(h(x) - h(y)) \wedge \neg P(0).$$

The first step is to construct three conjunctions $F_{\mathscr{E}}$, $F_{\mathscr{R}}$, and $F_{\mathscr{L}}$, such that $F_{\mathscr{E}}$ contains only $\mathscr{E}$-literals, $F_{\mathscr{R}}$ only $\mathscr{R}$-literals, $F_{\mathscr{L}}$ only $\mathscr{L}$-literals, and F is satisfiable if and only if $F_{\mathscr{E}} \wedge F_{\mathscr{R}} \wedge F_{\mathscr{L}}$ is. We do this by introducing new variables to replace terms of the wrong "type" and adding equalities defining these new variables. For instance, the second conjunct above would be an $\mathscr{R}$-literal except that it contains the term **car**(**cons**(0, x)), which is not an $\mathscr{R}$-term. We therefore replace **car**(**cons**(0, x)) by a new variable, say g_1, and add to the conjunction the equality g_1 = **car**(**cons**(0, x)) defining g_1. By continuing in this fashion we eventually obtain the following three conjunctions:

$F_{\mathscr{R}}$	$F_{\mathscr{E}}$	$F_{\mathscr{L}}$
$x \leq y$	$P(g_2)$ = **true**	g_1 = **car**(**cons**(g_5, x))
$y \leq x + g_1$	$P(g_5)$ = **false**	
$g_2 = g_3 - g_4$	$g_3 = h(x)$	
$g_5 = 0$	$g_4 = h(y)$	

These three conjunctions are given to the three satisfiability procedures $\mathscr{R}$, $\mathscr{E}$, and $\mathscr{L}$. Since each conjunction is satisfiable by itself, there must be interaction between the procedures for the unsatisfiability to be detected. The interaction takes a particular, restricted form: we require that each satisfiability procedure deduce and *propagate* to the other satisfiability procedures all equalities between variables entailed by the conjunction it is considering. For example, if $x \leq y$ and $y \leq x$ are given to $\mathscr{R}$, it must deduce and propagate to the other satisfiability procedures the fact that $x = y$. The other satisfiability procedures add $x = y$ to their conjunctions and the process continues.

In our example, neither $F_{\mathscr{R}}$ nor $F_{\mathscr{E}}$ entail any equalities between variables, but $F_{\mathscr{L}}$ entails $g_1 = g_5$. $\mathscr{L}$ propagates this equality. $\mathscr{R}$ uses this equality to deduce and propagate $x = y$. $\mathscr{E}$ then propagates $g_3 = g_4$. $\mathscr{R}$ then propagates $g_2 = g_5$. Now $\mathscr{E}$ has an inconsistent conjunction, and signals **unsatisfiable**. The following shows the literals received by the satisfiability procedures, and the propagated equalities listed in the order in which they were propagated.

$\mathscr{R}$	$\mathscr{E}$	$\mathscr{L}$
$x \leq y$	$P(g_2)$ = **true**	g_1 = **car**(**cons**(g_5, x))
$y \leq x + g_1$	$P(g_5)$ = **false**	$g_1 = g_5$
$g_2 = g_3 - g_4$	$g_3 = h(x)$	
$g_5 = 0$	$g_4 = h(y)$	
$x = y$	$g_3 = g_4$	
$g_2 = g_5$	**unsatisfiable**	

If one of the conjunctions $F_{\mathscr{R}}$, $F_{\mathscr{E}}$, or $F_{\mathscr{L}}$ becomes unsatisfiable as a result of equality propagation, the original conjunction must be unsatisfiable. For $\mathscr{R}$, $\mathscr{E}$, and $\mathscr{L}$, the converse holds as well: if the original conjunction is unsatisfiable, then one of the conjunctions $F_{\mathscr{R}}$, $F_{\mathscr{E}}$, or $F_{\mathscr{L}}$ will become unsatisfiable as a result of propagations of equalities between variables. For some other theories, such as $\mathscr{A}$, the converse does not hold. For such theories, "case splitting" is required. As an example of case splitting, suppose that the theories being combined are $\mathscr{A}$ and

$\mathscr{R}$ and that the conjunction to be shown unsatisfiable is

$$\textbf{store}(v, i, e)[j] = x \wedge v[j] = y \wedge x > e \wedge x > y.$$

As before, the formula is first divided into two conjunctions:

$$F_{\mathscr{A}}\text{: } \textbf{store}(v, i, e)[j] = x \wedge v[j] = y$$

$$F_{\mathscr{R}}\text{: } x > e \wedge x > y$$

Each formula is satisfiable, the whole conjunction is unsatisfiable, but there are no equalities to propagate. However, $F_{\mathscr{A}}$ entails the disjunction $x = e \vee x = y$. Since neither $x = e$ nor $x = y$ is consistent with $F_{\mathscr{R}}$, the original formula can be shown to be unsatisfiable by considering both cases.

Regardless of whether or not case splitting is required, it is never necessary to propagate "disequalities," nor equalities other than those between variables. For instance, in the first example there was no need for $\mathscr{R}$ to propagate $y \leq x$ or $x = y + g_5$ after receiving $g_1 = g_5$, although these were deducible facts. None of the other satisfiability procedures could make use of this information, since none of them knows anything about $\leq$ or $+$. Further, no disequality need be propagated, even though every theory shares $=$ and $\neg$. A disequality $x \neq y$ is needed to prove inconsistency only if $x = y$ is deduced. If some procedure deduces $x = y$, it will propagate this fact to the other procedures, and the one that has deduced $x \neq y$ will detect the inconsistency.

In Section 4 we specify precisely how equality propagation and case splitting can be used to combine decision procedures.

4. EQUALITY PROPAGATION PROCEDURE

If $\mathscr{S}$ and $\mathscr{T}$ are two theories with no common nonlogical symbols, their *combination* is the theory whose set of nonlogical symbols is the union of the sets of nonlogical symbols of $\mathscr{S}$ and $\mathscr{T}$, and whose set of axioms is the union of the sets of axioms of $\mathscr{S}$ and $\mathscr{T}$. We do not consider combining theories which share nonlogical symbols. Let F be a conjunction of literals whose nonlogical symbols are among those of $\mathscr{S}$ and $\mathscr{T}$. The following algorithm determines whether F is satisfiable in the combination of $\mathscr{S}$ and $\mathscr{T}$. The algorithm uses the variables $F_{\mathscr{S}}$ and $F_{\mathscr{T}}$ which contain conjunctions of literals.

Equality Propagation Procedure

1. Assign conjunctions to $F_{\mathscr{S}}$ and $F_{\mathscr{T}}$ by the method described in Section 3 so that $F_{\mathscr{S}}$ contains a conjunction of $\mathscr{S}$-literals, $F_{\mathscr{T}}$ a conjunction of $\mathscr{T}$-literals, and $F_{\mathscr{S}} \wedge F_{\mathscr{T}}$ is satisfiable if and only if F is.
2. [Unsatisfiable?] If either $F_{\mathscr{S}}$ or $F_{\mathscr{T}}$ is unsatisfiable, return **unsatisfiable**.
3. [Propagate equalities.] If either $F_{\mathscr{S}}$ or $F_{\mathscr{T}}$ entails some equality between variables not entailed by the other, then add the equality as a new conjunct to the one that does not entail it. Go to step 2.
4. [Case split necessary?] If either $F_{\mathscr{S}}$ or $F_{\mathscr{T}}$ entails a disjunction $u_1 = v_1 \vee \cdots \vee u_k = v_k$ of equalities between variables, without entailing any of the equalities alone, then apply the procedure recursively to the k formulas $F_{\mathscr{S}} \wedge F_{\mathscr{T}} \wedge u_1 = v_1, \ldots, F_{\mathscr{S}} \wedge F_{\mathscr{T}} \wedge u_k = v_k$. If any of these formulas are satisfiable, return **satisfiable**. Otherwise return **unsatisfiable**.
5. Return **satisfiable**.

If the procedure returns **unsatisfiable**, it is clear that F is unsatisfiable. We prove in Section 5 that the procedure is also correct if it returns **satisfiable**. The procedure always halts, since each repetition of step 3 or recursive call in step 4 conjoins an equality to one of the conjunctions $F_{\mathcal{S}}$ or $F_{\mathcal{T}}$ not previously entailed by the conjunction. This can happen at most $n - 1$ times, where n is the number of variables appearing after step 1, since there can be no more than $n - 1$ nonredundant equalities between n variables.

We call a formula F *nonconvex* if it would cause a case split; that is, if there exist $2n$ variables $x_1, y_1, \ldots, x_n, y_n$, such that F entails $x_1 = y_1 \vee \cdots \vee x_n = y_n$ but for no i between 1 and n does F entail $x_i = y_i$. Otherwise, F is *convex*. Step 4 causes a case split whenever $F_{\mathcal{S}}$ or $F_{\mathcal{T}}$ becomes nonconvex. A theory $\mathcal{S}$ is *convex* if every conjunction of $\mathcal{S}$-literals is convex. As we shall see, $\mathcal{R}$, $\mathcal{E}$, and $\mathcal{L}$ are convex, so they never cause case splitting. The example in Section 3 showed that $\mathcal{A}$ is not convex.

In theory, any decision procedures for $\mathcal{S}$ and $\mathcal{T}$ can be used to determine the equalities and disjunctions of equalities that need to be propagated in steps 3 and 4 simply by testing which of the finitely many equalities and disjunctions of equalities between variables are entailed. Much better methods are generally possible, and our satisfiability procedures $\mathcal{R}$, $\mathcal{E}$, $\mathcal{A}$, and $\mathcal{L}$ determine the propagated equalities and disjunctions rapidly, using different methods for each theory.

$\mathcal{R}$ uses the simplex algorithm to determine the satisfiability of conjunctions of linear inequalities. This algorithm is fast in practice and can be modified in a fairly straightforward manner so that it detects the equalities which are consequences of the linear inequalities. It is not difficult to see that $\mathcal{R}$ is convex: the solution set of a conjunction of linear inequalities is a convex set; the solution set of a disjunction of equalities is a finite union of hyperplanes; and a convex set cannot be contained in a finite union of hyperplanes unless it is contained in one of them.

In [7] we describe satisfiability procedures for $\mathcal{E}$ and $\mathcal{L}$ and prove that $\mathcal{E}$ and $\mathcal{L}$ are convex. Both these satisfiability procedures take time $O(n^2)$ to determine the satisfiability of, and the equalities entailed by, a conjunction of length n. Downey et al. [3] have improved the underlying algorithm to $O(n \log^2 n)$. Oppen [9] describes a satisfiability procedure for $\mathcal{L}$ which runs in linear time if list structure is assumed to be acyclic.

The satisfiability problem for conjunctions of $\mathcal{A}$-literals is NP complete [2]. The algorithm for $\mathcal{A}$ is not difficult (it just does the obvious case splits) but can be very costly.

Most of the theories we would like to add to our simplifier are nonconvex. The theory of the reals under multiplication is not convex; for example, $xy = 0 \wedge z = 0$ entails the disjunction $x = z \vee y = z$. Neither is the theory of the integers under $+$ and $\leq$. For example, $x = 1 \wedge y = 2 \wedge 1 \leq z \wedge z \leq 2$ entails the disjunction $x = z \vee y = z$. However, since we need only propagate equalities between variables, not between variables and constants, conjunctions such as $1 \leq z \leq 100$ will not cause hundred-way splits (unless there are 100 variables equal to 1, 2, ..., 100, respectively!). The theory of sets is wantonly nonconvex; for example, $\{a, b, c\} \cap \{c, d, e\} \neq \{\ \}$ forces a nine-way case split.

Oppen [8] shows how the complexity of a combination of theories is determined

by the complexity of the individual theories and their convexity. In particular, it is shown that the decision problem for the theory decided by our simplifier is NP complete.

5. CORRECTNESS OF THE EQUALITY PROPAGATION PROCEDURE

The proof of correctness requires several lemmas. Our first goal is to define the *residue* of a formula. Essentially the residue is the strongest Boolean combination of equalities between variables entailed by the formula. For example, the residue of the formula $x = f(a) \wedge y = f(b)$ is $a = b \supset x = y$, and the residue of $x \leq y \wedge y \leq x$ is $x = y$.

We assume that there are no propositional variables. This restriction is not essential, but it simplifies the proof.

A *parameter* of a formula is any nonlogical symbol which occurs free in the formula. Thus the parameters of $a = b \vee \forall x f(x) < c$ are a, b, f, $<$, and c.

We define a *simple* formula to be one whose only parameters are variables. For instance, $x \neq y \vee z = y$ and $\forall x\, x \neq y$ are simple, but $x < y$ and $f(x) = y$ are not. Thus an unquantified simple formula is a propositional formula whose atomic formulas are equalities between variables. Lemma 1 characterizes quantified simple formulas.

LEMMA 1. *Every quantified simple formula F is equivalent to some unquantified simple formula G. G can be chosen so that its variables are all free variables of F.*

PROOF. Suppose F is of the form $\exists x \Psi(x)$. Let Ψ_0 be the formula resulting from Ψ by first replacing any occurrences of $x = x$ and $x \neq x$ by **true** and **false**, respectively, and replacing any remaining equality involving x by **false**. Then, if $v_1, \ldots, v_k$ are the parameters of Ψ, F is equivalent to $\Psi_0 \vee \Psi(v_1) \vee \cdots \vee \Psi(v_k)$, since, in any interpretation, x either equals one of the v_i or differs from all of them. By repeatedly eliminating quantifiers in this manner, we eventually obtain an equivalent quantifier-free simple formula whose only variables are free variables of F.

LEMMA 2. (Craig's Interpolation Lemma). *If F and G are formulas such that F entails G, then there exists a formula H such that F entails H and H entails G, and each parameter of H is a parameter of both F and G.*

PROOF. See Craig [1] or Shoenfield [10].

LEMMA 3. *If F is any formula, then there exists a simple formula $Res(F)$, the residue of F, which is the strongest simple formula that F entails; that is, if H is any simple formula entailed by F, then $Res(F)$ entails H. $Res(F)$ can be written so that its only variables are free variables of F.*

PROOF. Let $\{G_\lambda\}$ be the set of all simple formulas entailed by F. For each G_λ, choose H_λ so that $F \supset H_\lambda \supset G_\lambda$, the only parameters of H_λ are parameters of both F and G_λ, and H_λ is unquantified. The existence of H_λ is guaranteed by Lemmas 1 and 2. Each H_λ is a propositional formula whose atomic formulas are equalities between parameters of F. It is easy to show that an infinite conjunction of propositional formulas over a finite set of atomic formulas is equivalent to some finite subconjunction. Therefore the conjunction of the H_λ is equivalent to some finite subconjunction H. Any simple formula G_λ entailed by F is entailed by some

H_λ, and so by H. The only parameters of H are free variables of F. Thus H is the residue of F.

Here are some examples of residues:

Formula	Residue
$x = f(a) \wedge y = f(b)$	$a = b \supset x = y$
$x + y - a - b > 0$	$\neg(x = a \wedge y = b) \wedge \neg(x = b \wedge y = a)$
$x = \mathbf{store}(v, i, e)[j]$	$i = j \supset x = e$
$x = \mathbf{store}(v, i, e)[j] \wedge y = v[j]$	**if** $i = j$ **then** $x = e$ **else** $x = y$

Notice in the last two formulas how the addition of a variable as a "label" affects the residue.

As a final example to relate the notion of residue to that of equality propagation, here are the residues of the formulas which appeared in the example of Section 3:

$\mathscr{R}$	$\mathscr{E}$	$\mathscr{L}$
$x \leq y$	$P(g_2)$	$g_1 = \mathbf{car}(\mathbf{cons}(g_5, x))$
$y \leq x + g_1$	$\neg P(g_5)$	$g_1 = g_5$
$g_2 = g_3 - g_4$	$g_3 = h(x)$	
$g_5 = 0$	$g_4 = h(y)$	
$g_5 = g_1 \equiv x = y \wedge$	$g_2 \neq g_5 \wedge$	
$g_3 = g_4 \equiv g_2 = g_5$	$x = y \supset g_3 = g_4$	

As we found in Section 3, the residues are inconsistent. An essential fact in the proof of correctness of the equality propagation procedure is that these residues are always inconsistent if the original formula is. This fact is a consequence of the following lemma.

LEMMA 4. *If A and B are formulas whose only common parameters are variables, then $Res(A \wedge B) \equiv Res(A) \wedge Res(B)$.*

Notice that the condition of the lemma is satisfied when A and B are from different theories which have no nonlogical symbols in common.

PROOF. Obviously the left side of the equivalence entails the right side, so we need only show the converse. From $A \wedge B \supset Res(A \wedge B)$ follows $A \supset (B \supset Res(A \wedge B))$. So, by Craig's interpolation lemma, there is a formula H entailed by A which entails $B \supset Res(A \wedge B)$, and whose only parameters are parameters of A and B. These must be variables, so H is simple. Therefore $Res(A) \supset (B \supset Res(A \wedge B))$. Writing this as $B \supset (Res(A) \supset Res(A \wedge B))$, and observing that the right-hand side is simple, we deduce that $Res(B) \supset (Res(A) \supset Res(A \wedge B))$, or, equivalently, that $Res(A) \wedge Res(B) \supset Res(A \wedge B)$, which proves the lemma.

LEMMA 5. *Let F and G be simple, satisfiable, convex formulas, and let V be the set of all variables appearing in either F or G. Suppose that for all x and y in V, either both F and G entail $x = y$ or neither do. Then $F \wedge G$ is satisfiable.*

PROOF. Let S be the set of equalities between variables in V entailed by F (hence also by G), and let T be the set of all other equalities between variables of V. We claim that any interpretation which makes every equality in S true and every equality in T false satisfies both F and G. If it does not satisfy, say, F, then F entails the disjunction of all equalities in T. We now consider three cases: If T is empty, F is unsatisfiable. If T contains only one equality, it is entailed by F and so it is in S. If T contains more than one equality, F is nonconvex. Each case contradicts our assumptions.

We now complete the proof of correctness of the equality propagation procedure by showing that if it returns **satisfiable**, F is satisfiable. Suppose that $F_{\mathcal{S}} \wedge F_{\mathcal{T}}$ is unsatisfiable. Then there is a finite conjunction S of axioms of $\mathcal{S}$ and a finite conjunction T of axioms of $\mathcal{T}$ such that $S \wedge F_{\mathcal{S}} \wedge T \wedge F_{\mathcal{T}}$ is unsatisfiable. Thus $Res(S \wedge F_{\mathcal{S}} \wedge T \wedge F_{\mathcal{T}})$ is the constant **false**. From Lemma 4, it follows that the conjunction of $Res(S \wedge F_{\mathcal{S}})$ with $Res(T \wedge F_{\mathcal{T}})$ is unsatisfiable. But if step 5 of the equality propagation procedure is reached, each of these residues must be convex, since step 4 did not cause a case split. Furthermore, the residues entail the same set of equalities and are each satisfiable, since steps 2 and 3 were passed. Lemma 5 implies that the conjunction of the residues is satisfiable, contrary to our assumption. Thus F is satisfiable if the algorithm returns from step 5. It follows, by induction on the depth of recursion, that F is satisfiable whenever step 4 returns **satisfiable**.

6. SIMPLIFYING BOOLEAN STRUCTURE

Simplifying a formula is more difficult than just determining whether it is satisfiable. If a formula is valid, its simplest equivalent is **true**; if it is unsatisfiable, its simplest equivalent is **false**; but if it is neither, the choice of the "simplest" form is a matter of taste. For example, $A \wedge (B \vee C)$ and $A \wedge B \vee A \wedge C$ are equivalent, but it is not easy to decide which is simpler. When an expression does not simplify to a constant such as **true**, our simplifier returns a variant of *cond normal form* (see McCarthy [6]), although this is not necessarily the most satisfactory form.

We assume in this section that the expression given to the simplifier is Boolean valued.

The simplifier first replaces $\wedge$, $\vee$, $\neg$, and $\supset$ by **if-then-else** using the rules:

$$\begin{aligned} a \wedge b &\rightarrow \textbf{if } a \textbf{ then } b \textbf{ else false} \\ a \vee b &\rightarrow \textbf{if } a \textbf{ then true else } b \\ a \supset b &\rightarrow \textbf{if } a \textbf{ then } b \textbf{ else true} \\ \neg a \quad &\rightarrow \textbf{if } a \textbf{ then false else true} \end{aligned} \tag{2}$$

It also uses the following transformation recursively until the first argument of a conditional expression is not itself a proper conditional expression:

$$\begin{aligned} &\textbf{if (if } p \textbf{ then } a \textbf{ else } b) \textbf{ then } c \textbf{ else } d \\ &\qquad \rightarrow \qquad \textbf{if } p \textbf{ then (if } a \textbf{ then } c \textbf{ else } d) \textbf{ else (if } b \textbf{ then } c \textbf{ else } d). \end{aligned}$$

(This transformation may double the "print length" of the formula, since c and d each occur once on the left and twice on the right. But in a conventional list-structure representation, the two occurrences can share storage.)

Finally, each atomic formula p which is not the test of an **if-then-else** is replaced by **if** p **then true else false**.

The cost of normalizing the formula by these transformations is linear in time and space.

Once the formula has been normalized, the simplifier "symbolically evaluates" the resulting conditional expression. The following function *simplify*(f) is a simplified version of the symbolic evaluator we use. It uses a routine *simpatom* which simplifies atomic formulas; for example, $simpatom(P(x + 0)) = P(x)$.

simplify(f):

1. If f is **true** or **false**, then return f. Otherwise, f is of the form if p **then** a **else** b, where p is an atomic formula. Set $p \leftarrow$ *simpatom*(p).
2. Assume p is true. If this assumption is inconsistent with the existing context of assumptions, then remove this assumption and return *simplify*(b). Otherwise, set $a \leftarrow$ *simplify*(a) and remove the assumption that p is true.
3. Assume p is false. If this assumption is inconsistent with the existing context of assumptions, then remove this assumption and return a. Otherwise, set $b \leftarrow$ *simplify*(b) and remove the assumption that p is false.
4. If a and b are identical, return a. If a is **true** and b is **false**, then return p. If the expression **if** p **then** a **else** b matches one of the right-hand sides of (2), then return the corresponding left-hand side. Otherwise, return **if** p **then** a **else** b.

For example, to simplify $p \supset \neg p$, the simplifier first normalizes it to the conditional expression

if p **then (if** p **then false else true) else true.**

This expression is symbolically evaluated. Under the assumption that p is **true**, the subexpression **if** p **then false else true** is recursively simplified to **false**. The other branch (the **else** branch) remains **true**, so the whole expression simplifies to **if** p **then false else true**, hence, by step 4, to $\neg p$.

To simplify $x = x$, the simplifier first normalizes it to **if** $x = x$ **then true else false**. The assumption $x \neq x$ is inconsistent, so the expression is simplified to **true**.

The equality propagation procedure is used to determine at any stage whether the conjunction of assumptions is satisfiable. To make this practical, the satisfiability procedures we use have two properties in addition to their ability to determine the satisfiability of conjunctions and to propagate the equalities and disjunctions of equalities entailed by the conjunctions. First, they are *incremental*, that is, they accept literals one by one, maintain a representation of their conjunction, and detect unsatisfiability of this conjunction as soon as it occurs. Second, they are *resettable*, that is, they can mark their state, accept further literals, and then return to the marked state by removing the literals received after the mark.

The algorithms we use for $\mathscr{R}$, $\mathscr{E}$, $\mathscr{L}$, and $\mathscr{A}$ are incremental and resettable. The data structure for the simplex algorithm used in $\mathscr{R}$ is a sparse matrix; adding and removing assumptions from $\mathscr{R}$'s conjunction is achieved by adding and removing rows of the matrix. Our algorithms for $\mathscr{E}$ and $\mathscr{L}$ are also incremental and resettable; given a conjunction of length n, the total time required to assume the literals one by one and then remove the assumptions one by one is $O(n^2)$.

The definition of *simplify* given above does not treat case splitting. Case splitting is necessary whenever the conjunction in one of the satisfiability procedures becomes nonconvex. If this happens, the rest of the subformula is recursively simplified once for each branch of the case split. For example, consider the formula

$$\textbf{store}(v, i, e)[j] = x \wedge v[j] = f \wedge (x \leq e \vee x \leq f),$$

which is normalized to

if store$(v, i, e)[j] = x$
then if $v[j] = f$ **then (if** $x \leq e$ **then true else** $x \leq f$**) else false**
else false.

When $v[j] = f$ is assumed, the conjunction in $\mathscr{A}$ becomes nonconvex, and $\mathscr{A}$ signals the split $x = e \vee x = f$. Instead of going on to evaluate **if** $x \leq e$ **then true else** $x \leq f$, the simplifier evaluates the expression:

$$\begin{aligned}&\textbf{if } x = e \textbf{ then } (\textbf{if } x \leq e \textbf{ then true else } x \leq f)\\ &\qquad\textbf{else } (\textbf{if } x \leq e \textbf{ then true else } x \leq f).\end{aligned}$$

Thus *simplify* is called twice with the argument (**if** $x \leq e$ **then true else** $x \leq f$), once under the assumption $x = e$ and once under the assumption $x \neq e$. When $x \neq e$ is assumed, $\mathscr{A}$ propagates $x = f$, thus effecting the case split. Both calls return **true**, so the final result is

$$\textbf{store}(v, i, e)[j] = x \wedge v[j] = f.$$

ACKNOWLEDGMENT

We thank the members of the Stanford Verification Group for their careful criticism of earlier drafts of this paper and of earlier versions of the simplifier.

REFERENCES

1. CRAIG, W. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic 22* (1955).
2. DOWNEY, P.J., AND SETHI, R. Assignment commands with array references. *J. ACM 25*, 4 (Oct. 1978), 652–666.
3. DOWNEY, P., SETHI, R., AND TARJAN, R. Variations on the common subexpression problem. To appear in *J. ACM.*
4. KAPLAN, D.M. Some completeness results in the mathematical theory of computation. *J. ACM 15*, 1 (Jan. 1968), 124–134.
5. MATIYASEVICH, Y.V. Diophantine representation of recursively enumerable predicates. Proc. 2nd Scandinavian Logic Symposium, North-Holland Publ. Co., 1970.
6. MCCARTHY, J. A basis for a mathematical theory of computation. In *Computing Programming and Formal Systems*, P. Braffort and D. Hirshberg, Eds. North-Holland Publ. Co., 1963.
7. NELSON, C.G., AND OPPEN, D.C. Fast decision algorithms based on congruence closure. CS Rep. STAN-CS-77-646, Stanford U., 1978.
8. OPPEN, D.C. Convexity, complexity, and combinations of theories. To appear in *Theoretical Comptr. Sci.*
9. OPPEN, D.C. Reasoning about recursively defined data structures. To appear in *J. ACM.*
10. SCHOENFIELD, J.R. *Mathematical Logic.* Addison-Wesley, Reading, Mass., 1967.
11. SHOSTAK, R.E. An practical decision procedure for arithmetic with function symbols. *J. ACM 26*, 2 (April 1979), 351–360.
12. SUZUKI, N., AND JEFFERSON, D. Verification decidability of Presburger array programs. Proc. Conf. on Theoretical Computer Science, U. of Waterloo, Aug. 1977.
13. TAKEUCHI, I. Private communication to J. McCarthy.
14. TARSKI, A. A decision method for elementary algebra and geometry. Berkeley, 1951.

Received April 1978; revised January 1979