A SIMPLIFIER BASED ON EFFICIENT DECISION ALGORITHMS

Greg Nelson
and
Derek C. Oppen

Artificial Intelligence Laboratory
Computer Science Department
Stanford University
Stanford, California

Abstract

We describe a simplifier for use in program manipulation and verification. The simplifier finds a normal form for any expression over the language consisting of individual variables, the usual boolean connectives, the conditional function **cond** (denoting **if-then-else**), the integers (numerals), the arithmetic functions and predicates +, - and ≤, the LISP constants, functions and predicates **nil, car, cdr, cons** and **atom,** the functions **store** and **select** for storing into and selecting from arrays, and uninterpreted function symbols. Individual variables range over the union of the rationals, the set of arrays, the LISP s-expressions and the booleans **true** and **false.** The constant, function and predicate symbols take their natural interpretations.

The simplifier is *complete*; that is, it simplifies every valid formula to **true.** Thus it is also a decision procedure for the quantifier-free theory of rationals, arrays and s-expressions under the above functions and predicates.

The organization of the simplifier is based on a method for combining decision algorithms for several theories into a single decision algorithm for a larger theory containing the original theories. More precisely, given a set S of functions and predicates over a fixed domain, a *satisfiability program* for S is a program which determines the satisfiability of conjunctions of literals (signed atomic formulas) whose predicates and function signs are in S. We give a general procedure for combining satisfiability programs for sets S and T into a single satisfiability program for S ∪ T, given certain conditions on S and T. We show how a satisfiability program for a set S can be used to write a complete simplifier for expressions containing functions and predicates of S as well as uninterpreted function symbols.

The simplifier described in this paper is currently used in the Stanford Pascal Verifier.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206 and by the National Science Foundation under contract MCS 76-000327.

1. Introduction

In this section we will define the syntax and semantics of the language accepted by our simplifier, and give some examples of simplifications. We will also define a *satisfiability program* for a set S of functions, predicates, and constants. Essentially, such a program determines the satisfiability of conjunctions of literals (signed atomic formulas) whose predicates and function signs are in S. The formal definition specifies the interpretations of the elements of S in such a way that it makes sense to "merge" satisfiability procedures for two sets S and T into one for S ∪ T. Section 2 gives our method for doing this, based on Craig's interpolation lemma ([Craig 1957]). Section 3 shows how a satisfiability procedure can be used to implement a simplifier for arbitrary expressions. Section 4 contains some concluding remarks.

1.1 Language Accepted by the simplifier

The simplifier accepts the usual boolean connectives and also the three-argument function **cond** (**cond**(p,a,b) means **if** p **then** a **else** b). The other functions, predicates, and constants to which the simplifier currently gives an interpretation are those of the following theories: the theory of rationals under addition, the theory of list structure with **car, cdr, cons, atom** and **nil,** and the theory of arrays under storing (**store**) and selecting (**select**). The following axioms are assumed for these functions.

Arithmetic: $\mathcal{Z}$

$$
\begin{aligned}
&\forall x\ \forall y\ \forall z\\
&[x + 0 = x\\
&\wedge\ x + -x = 0\\
&\wedge\ (x + y) + z = x + (y + z)\\
&\wedge\ x + y = y + x\\
&\wedge\ x \le x\\
&\wedge\ (x \le y \vee y \le x)\\
&\wedge\ (x \le y \wedge y \le x \supset x = y)\\
&\wedge\ (x \le y \wedge y \le z \supset x \le z)\\
&\wedge\ (x \le y \supset x + z \le y + z)\\
&\wedge\ 0 \neq 1\\
&\wedge\ 0 \le 1]
\end{aligned}
$$

We also allow the other arithmetic relations <, >, and ≥, multiplication by a scalar, and the numerals 2, 3, ... These can be defined in terms of 0, +, - and ≤ in the usual way. The above axiom is the axiom for a commutative group with a translation-invariant total order. A model for the theory with this axiom is the theory of rationals under addition and ≤. ($\mathcal{Z}$ is called $\mathcal{Z}$ instead of Q for historical reasons.)

Arrays: $\mathcal{V}$

∀v ∀e ∀i ∀j
select(store(v, i, e), j) = cond(i=j, e, select(v, j))

The simplifier accepts **store** and **select**, functions which operate on arrays or vectors. select(V,I) denotes the Ith component of the vector V. We may write V[I] for select(V,I). store(V,I,E) is the vector whose Ith component is equal to E and whose Jth component, for J ≠ I, is the Jth component of V.

List Structure: $\mathcal{L}$

∀x ∀y
[**car(cons(X, Y))** = X
∧ **cdr(cons(X, Y))** = Y
∧ ¬ **atom(X)** ⊃ **cons(car(X), cdr(X))** = X
∧ ¬ **atom(cons(X, Y))**
∧ **atom(nil)**]

Note that acyclicity is not assumed; for instance, **car**(X) = X is regarded as satisfiable.

Let $\mathcal{A}$ be the conjunction of the above axioms. Given a quantifier-free expression F, the simplifier tries to find the simplest F' such that $\mathcal{A}$ entails F = F' and returns F'. In particular, if $\mathcal{A}$ entails F, the simplest F' such that $\mathcal{A}$ entails F = F' is the boolean constant **true**, which the simplifier will return. Thus the simplifier is a decision procedure for the quantifier-free theory axiomatized by $\mathcal{A}$.

The results in this paper apply to many logical theories, but we will illustrate them with the theories which our simplifier currently handles. We use $\mathcal{Z}$, $\mathcal{V}$, and $\mathcal{L}$ as names for the theories of rationals, arrays and list structure. We also use these letters for the axioms for these theories (note that each theory has exactly one axiom) and for their satisfiability programs.

Besides the satisfiability programs $\mathcal{Z}$, $\mathcal{L}$, and $\mathcal{V}$, our simplifier contains a program, called $\mathcal{E}$, which determines the satisfiability of conjunctions of equalities and disequalities between terms containing uninterpreted functions and predicates. For instance, $\mathcal{E}$ will return **unsatisfiable** given the formula x = y ∧ f(x) ≠ f(y). $\mathcal{E}$ is therefore a satisfiability program for the quantifier-free theory of equality with uninterpreted function symbols.

1.2 Examples of the Use of the Simplifier

Here are some examples of simplifications.

P ⊃ ¬ P;
¬ P;

X = F(X) ⊃ F(F(F(X))) = X;
true;

cons(X,Y) = Z ⊃ **car**(Z) + **cdr**(Z) - X - Y = 0;
true;

X ≤ Y ∧ Y + D ≤ X ∧ 3 * D ≥ 2 * D ⊃ V[2 * X - Y] = V[X + D];
true;

V = **store**(**store**(V, I, V[J]), J, V[I]) ⊃ V[I] = V[J];
true;

This formula expresses the theorem that if the Ith and Jth elements of a vector V are swapped, and if the resulting vector is identical to the original vector, then the Ith and Jth elements were the same.

1.3 Satisfiability Programs

The *logical symbols* are ∧, ∨, ¬, ⊃, =, **cond**, ∀ and ∃. A *parameter* of a formula is any non-logical atomic symbol which occurs free in the formula. Thus the parameters of the formula Q ∨ ∀x P(x, f(x)) = a are Q, P, f, and a. The parameters of the axiom for a theory are the primitive constants, functions, and predicates of the theory. For instance, the parameters of $\mathcal{L}$ are **car**, **cdr**, **cons**, **atom** and **nil**.

If $\mathcal{A}$ is an axiom, then a term is an *$\mathcal{A}$-term* if and only if each of its parameters is a parameter of $\mathcal{A}$ or an individual variable. We define *$\mathcal{A}$-literal* and *$\mathcal{A}$-formula* analogously. For example, x = y and x ≤ y + 3 are $\mathcal{Z}$-literals but x ≤ **car**(y) is not.

We call a term an *$\mathcal{E}$-term* if none of its parameters are parameters of any axiom, that is, if all its parameters are uninterpreted. We similarly define *$\mathcal{E}$-literal* and *$\mathcal{E}$-formula*. (Note that this definition only makes sense if we have already fixed the other theories we are interested in; we will assume we have fixed $\mathcal{Z}$, $\mathcal{L}$ and $\mathcal{V}$.)

A *satisfiability program* for $\mathcal{A}$ is a program which determines whether a conjunction $L_1 \wedge \ldots \wedge L_k$ of $\mathcal{A}$-literals is satisfiable in the theory axiomatized by $\mathcal{A}$, that is, whether $\mathcal{A} \wedge L_1 \wedge \ldots \wedge L_k$ is satisfiable. $\mathcal{A}$ specifies a set of functions and predicates and also their interpretation; this makes precise the definition given in the introduction.

There are efficient satisfiability programs for $\mathcal{Z}$, $\mathcal{V}$ and $\mathcal{L}$. For $\mathcal{Z}$, the simplex algorithm is very fast in practice ([Nelson 1976]). [Nelson and Oppen 1977] describe satisfiability algorithms for $\mathcal{L}$ and $\mathcal{E}$ which determine the satisfiability of conjunctions of length n in time $O(n^2)$. [Johnson and Tarjan 1977] have improved the underlying algorithm to $O(n \log^2 n)$. [Oppen 1978]

describes an algorithm which runs in linear time if list structure is assumed to be acyclic. The satisfiability problem for $\mathcal{V}$ is NP-complete [Downey and Sethi 1976], but the expensive cases do not seem to arise in practice.

2. Merging Satisfiability Programs

In this section, we show how to write a satisfiability program for, say, $\mathcal{Z} \wedge \mathcal{L}$, given one for $\mathcal{Z}$ and one for $\mathcal{L}$. It is not obvious how to do this since there may be "mixed" terms in the conjunction like car(x) ≤ cdr(x) + 1. Our method for merging satisfiability programs for two theories will work whenever their sets of parameters are disjoint. This is the case for $\mathcal{Z}$, $\mathcal{V}$ and $\mathcal{L}$.

We will also show how a satisfiability program for a theory can be used in conjunction with $\mathcal{E}$ to decide the satisfiability of conjunctions of literals which contain uninterpreted function symbols as well as parameters of the theory.

We will first illustrate how we merge satisfiability programs.

2.1 Example

Let F be the following unsatisfiable conjunction:

$$X \le Y \wedge Y \le X + \mathbf{car}(\mathbf{cons}(0,X)) \wedge P(F(X)\text{-}F(Y)) \wedge \neg P(0);$$

The first step is to make each atomic formula *homogeneous*, that is, contain only parameters of one theory. 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 would be a $\mathcal{Z}$-literal except that it contains a term **car**(**cons**(0,X)), which is not a $\mathcal{Z}$-term. We replace **car**(**cons**(0,X)) by a new variable G1, say, and add to the conjunction the equality G1 = **car**(**cons**(0,X)) defining G1. By continuing in this fashion we eventually obtain a formula F' which is satisfiable if and only if F is, such that each literal of F' is homogeneous. In our example, F' is

$$X \le Y \wedge Y \le X + G1 \wedge P(G2) \wedge \neg P(G5) \wedge G1 = \mathbf{car}(\mathbf{cons}(G5,X)) \wedge G2 = G3 - G4 \wedge G3 = F(X) \wedge G4 = F(Y) \wedge G5 = 0;$$

We next divide F' up into three conjunctions F_E, F_Z and F_L. F_E contains all the $\mathcal{E}$-literals, F_Z all the $\mathcal{Z}$-literals and F_L all the $\mathcal{L}$-literals. Here is F' divided up into homogeneous parts:

F_Z	F_E	F_L
X ≤ Y	P(G2) = **true**	G1 = **car**(**cons**(G5,X))
Y ≤ X + G1	P(G5) = **false**	
G2 = G3 - G4	G3 = F(X)	
G5 = 0	G4 = F(Y)	

These three conjunctions are given to the three satisfiability programs $\mathcal{Z}$, $\mathcal{E}$, and $\mathcal{L}$. Since each conjunction is satisfiable by itself, there must be interaction between the programs to detect the unsatisfiability. The interaction takes a particular, restricted form. We require that each satisfiability program deduce and *propagate* to the other satisfiability programs all (new) equalities between variables entailed by the conjunction it is considering. For example, if X ≤ Y and Y ≤ X are asserted to $\mathcal{Z}$, it must deduce and propagate to the other satisfiability programs the fact that X = Y. The other satisfiability programs add X = Y to their conjunctions and the process continues. Eventually, either some satisfiability program will find that the conjunction it has received is unsatisfiable (in which case the original formula was unsatisfiable) or else the propagations will stop with each conjunction satisfiable. In this case, one more test, described in the next section, must be made to guarantee that the original formula is satisfiable. (In our example, this test is not needed.)

Let us illustrate this by continuing with our example. The satisfiability programs $\mathcal{Z}$, $\mathcal{E}$ and $\mathcal{L}$ have just received, respectively, F_Z, F_E and F_L. ($\mathcal{V}$ is not needed since there are no $\mathcal{V}$-terms.)

At first, only $\mathcal{L}$ can deduce an equality between variables: the equality G1 = G5. It propagates this equality. $\mathcal{Z}$ can make use of this fact and propagates X = Y. $\mathcal{E}$ now propagates G3 = G4. When $\mathcal{Z}$ receives this new equality, it propagates G2 = G5. $\mathcal{E}$ now has an inconsistent conjunction, and signals **unsatisfiable**.

The following shows the literals received by the satisfiability programs, together with the propagated equalities listed in the order in which they were propagated.

$\mathcal{Z}$	$\mathcal{E}$	$\mathcal{L}$
X ≤ Y	P(G2) = **true**	G1 = **car**(**cons**(G5,X))
Y ≤ X + G1	P(G5) = **false**	
G2 = G3 - G4	G3 = F(X)	
G5 = 0	G4 = F(Y)	
		G1 = G5
X = Y		
	G3 = G4	
G2 = G5		
	unsatisfiable	

There are several important observations to make.

First, the only interactions that take place between satisfiability programs is by propagations of equalities between variables. We prove in section 2.3 that this is sufficient for completeness (it is sometimes necessary to do a "case-split" as described in section 2.2).

It is important to realize that it is never necessary to propagate disequalities, nor equalities other than those between variables. This is intuitively plausible. For instance, after receiving G1 = G5, there was no need for $\mathcal{Z}$ to propagate that Y ≤ X or that X = Y + G5, even though these were deducible facts, since none of the other satisfiability programs could make use of this information – none of them knows anything about ≤ or +. Further, it is plausible that no disequality, such as x ≠ y, need be propagated, even though every theory shares = and ¬: the disequality x ≠ y is needed to prove inconsistency only if x = y is deduced. But if some program deduces that x = y, it will propagate

this fact to the other programs, in particular the one that has deduced $x \neq y$; the latter will deduce the inconsistency.

Finally, the only satisfiability programs that made use of a new equality, such as $x = y$, were those programs whose conjunctions contained occurrences of both x and y. For instance, when $\mathcal{L}$ propagated G1 = G5, only $\mathcal{Z}$ ever made direct use of this equality. It is in fact the case that when equalities are propagated, the only satisfiability programs that need to receive the equality are those which already "know" about both variables in the equality.

2.2 Joint Satisfiability Procedure

In this section we present a *joint satisfiability procedure* which combines satisfiability algorithms for several theories. We assume that we have just two theories $\mathcal{A}$ and $\mathcal{B}$ with no common parameters (the general case follows easily) and that we have satisfiability procedures for determining if $\mathcal{A}$-literals, $\mathcal{B}$-literals and $\mathcal{E}$-literals are satisfiable.

Given a conjunction F of literals, the joint satisfiability procedure determines whether $\mathcal{A} \wedge \mathcal{B} \wedge F$ is **satisfiable**. F_A, F_B and F_E are program variables containing conjunctions of literals.

1. [Make F homogeneous] Assign conjunctions to F_E, F_A, and F_B by the method described in section 2.1, so that F_E contains an E-formula, F_A an $\mathcal{A}$-formula, F_B a $\mathcal{B}$-formula, and $F_E \wedge F_A \wedge F_B \wedge \mathcal{A} \wedge \mathcal{B}$ is satisfiable iff $F \wedge \mathcal{A} \wedge \mathcal{B}$ is.

2. [Unsatisfiable?] If any of F_E, $\mathcal{A} \wedge F_A$, $\mathcal{B} \wedge F_B$ are unsatisfiable, return **unsatisfiable**.

3. [Propagate equalities] If any of the formulas F_E, $\mathcal{A} \wedge F_A$, $\mathcal{B} \wedge F_B$ entail some equality between variables which is not entailed by both of the other formulas, then add the equality as a new conjunct to whichever of F_A, F_B and F_E do not entail it. Go to 2.

4. [Case split necessary] If any of the formulas F_E, $\mathcal{A} \wedge F_A$, $\mathcal{B} \wedge F_B$ entail a disjunction $u_1 = v_1 \vee \ldots \vee u_k = v_k$ of equalities between variables, without entailing any of the equalities alone, then apply the joint satisfiability procedure recursively to the k formulas $F_A \wedge F_B \wedge F_E \wedge u_1 = v_1$, ..., $F_A \wedge F_B \wedge F_E \wedge u_k = v_k$. If any of these formulas are satisfiable, return **satisfiable**. Otherwise return **unsatisfiable**.

5. (If this step is reached, there are no equalities to propagate and no case splits to be done, and F_E, F_A, F_B are each satisfiable.) Return **satisfiable**.

Clearly if the procedure returns **unsatisfiable**, then F is unsatisfiable. We will prove in the next section that the procedure is also correct if it returns **satisfiable**.

The joint satisfiability procedure we have described here is too crude to be implemented; it will be subsumed by the simplification algorithm described in section 3.1.

We conclude this section with an example, involving $\mathcal{V}$ and $\mathcal{Z}$, which illustrates case splitting. Suppose that after step 1, the formulas are:

F_V: store(v,i,e)[j] = x $\wedge$ v[j] = y

F_Z: x > e $\wedge$ x > y

F_E: **true**

Each formula is satisfiable and the whole conjunction is unsatisfiable, but there are no equalities to propagate in step 3. In step 4, $\mathcal{V}$ propagates the disjunction $x = e \vee x = y$; each case leads to a contradiction in $\mathcal{Z}$.

As this example shows, case splitting is essential to the completeness of the method. It is potentially very expensive, but we have found that it does not occur frequently. In fact, $\mathcal{V}$ is the only satisfiability procedure in our system which can cause a split. $\mathcal{Z}$ cannot, since a conjunction of linear inequalities in rationals can never entail a disjunction of equalities, unless it entails one of the disjuncts. (Otherwise we would have a convex set contained in the union of two hyperplanes but not in either of them alone, which is impossible.) We can also prove that $\mathcal{E}$ and $\mathcal{L}$ never produce splits. We call theories which never produce splits *convex*.

When $\mathcal{Z}$ is extended to be a satisfiability program for the integers, it will no longer be convex, since for example $x = 1 \wedge y = 2 \wedge 1 \leq z \wedge z \leq 2$ entails the disjunction $x = z \vee y = z$ without entailing either disjunct alone. However, since we need propagate only equalities between variables, not between variables and constants, literals such as $1 \leq z \leq 100$ will not cause splits (unless there are 100 variables equal to 1, 2, ..., 100 respectively!).

The theory of sets, which we intend to add to the simplifier, is another example of a non-convex theory; for example, $x \in \{y, z\}$ causes a case split.

2.3 Correctness of the Joint Satisfiability 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 which the formula entails. 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 make the following assumptions about the underlying formal system: (1) Individual variables are distinguishable from function variables, and propositional variables are distinguishable from other individual variables, and (2) There is no quantification over functions or predicates. The results of this section hold without these restrictions, but the proofs are easier using (2) and less tedious using (1).

We define a *simple* formula to be one whose only parameters are individual variables. For instance, $x \neq y \vee z = y$ is simple, but $x < y$ is not. Thus an unquantified simple formula is a propositional formula whose atomic formulas are either

propositional variables or equalities between individual variables. Because of the next lemma, this also characterizes arbitrary simple formulas.

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

Proof: Suppose F is of the form $\exists x\ \Psi(x)$. If x is a propositional variable, we can take $\Psi(\mathbf{true}) \vee \Psi(\mathbf{false})$ as G. Otherwise, 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 then replacing any remaining equality involving x by **false**. Then, if $v_1, \ldots, v_k$ are the parameters of Ψ, F is equivalent to $\Psi_0 \vee \Psi(v_1) \vee \ldots \vee \Psi(v_k)$, since, in any interpretation, x will either equal one of the v_i or else be different from all of them. By repeatedly eliminating quantifiers in this manner, we will eventually obtain an equivalent quantifier-free formula whose only variables are free variables of F.

(If restriction (2) above is lifted, Lemma 1 holds only if we allow simple formulas to contain equalities between function variables. For example, $\forall x\ f(x) = g(x)$ is a quantified simple formula equivalent to the simple formula $f = g$.)

Lemma 2: (Craig's interpolation lemma) If 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, 1957].

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 individual variables of F.

Proof: Let $\{G_\lambda\}$ be the set of all simple formulas which F entails. 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. Now, each H_λ is a propositional formula whose atomic formulas are propositional variables which are parameters of F or else equalities between individual parameters of F. An infinite conjunction of propositional formulas over a finite set of atomic formulas is always equivalent to some finite propositional formula over these atomic formulas, so there is a finite formula H which is equivalent to the conjunction of all the H_λ, whose only parameters are free individual parameters of F. But any simple formula G_λ entailed by F is entailed by some H_λ, and so by H. We can therefore take H to be the residue of F.

If $\mathcal{A}$ is an axiom, the *$\mathcal{A}$-residue* of F is the residue of $\mathcal{A} \wedge F$. Thus the $\mathcal{Z}$-residue of $X \leq Y \wedge Y \leq X$ is $X = Y$. When no confusion can result, we will not specify the axiom.

Here are some examples of residues.

$\mathrm{Res}(x=f(a) \wedge y=f(b)) \equiv a=b \supset x=y$
$\mathrm{Res}(x=\mathbf{store}(v, i, e)[j]) \equiv i=j \supset x=e$
$\mathrm{Res}(x=\mathbf{store}(v, i, e)[j] \wedge y=v[j]) \equiv \mathbf{cond}(i=j, x=e, x=y).$

(Note that the addition of an individual variable as a "label" affects the residue.)

$\mathrm{Res}(x + y - a - b > 0) \equiv \neg(x=a \wedge y=b) \wedge \neg (x=b \wedge y=a)$

As a final example to relate the notion of residue to that of joint satifiability, here are the residues of the formulas which appeared in the example of section 2.1:

$\mathcal{Z}$	$\mathcal{E}$	$\mathcal{L}$
$X \leq Y$	$P(G2)$	$G1 = \mathbf{car}(\mathbf{cons}(G5,X))$
$Y \leq X + G1$	$\neg P(G5)$	
$G2 = G3 - G4$	$G3 = F(X)$	
$G5 = 0$	$G4 = F(Y)$	

$G5=G1 \equiv X=Y \wedge G3=G4 \equiv G2=G5$
$G2 \neq G5 \wedge X = Y \supset G3 = G4$
$G1 = G5$

As we found in section 2.1, the residues are inconsistent. An essential fact needed for proving the correctness of the joint satisfiability procedure is that these residues are always inconsistent if the original formula is. This is a consequence of the following lemma.

Lemma 4: If A and B are formulas whose only common parameters are individual variables, then $\mathrm{Res}(A \wedge B) \equiv \mathrm{Res}(A) \wedge \mathrm{Res}(B)$.

Proof: Obviously the left side of the equivalence entails the right side, so we need only show the converse.

From $A \wedge B \supset \mathrm{Res}(A \wedge B)$ we get $A \supset (B \supset \mathrm{Res}(A \wedge B))$ and so, by Craig's interpolation lemma, there is a formula H entailed by A which entails $B \supset \mathrm{Res}(A \wedge B)$ and whose only parameters are parameters of A and B. But these must be individual variables, so H is simple and therefore $\mathrm{Res}(A) \supset (B \supset \mathrm{Res}(A \wedge B))$. Writing this as $B \supset (\mathrm{Res}(A) \supset \mathrm{Res}(A \wedge B))$, and observing that the right hand side of this is simple, we have $\mathrm{Res}(B) \supset (\mathrm{Res}(A) \supset \mathrm{Res}(A \wedge B))$, or $\mathrm{Res}(A) \wedge \mathrm{Res}(B) \supset \mathrm{Res}(A \wedge B)$, which proves the lemma.

Now we are ready to prove the correctness of the Joint Satisfiability Procedure.

We will prove that if step 5 is reached, F is satisfiable. It follows, by induction on the depth of recursion, that the procedure is also correct whenever step 4 returns **satisfiable.**

To show that F is satisfiable, it suffices to show that $\mathrm{Res}(F_E \wedge A \wedge F_A \wedge B \wedge F_B)$ is not false. But by lemma 4, this is equivalent to $\mathrm{Res}(F_E) \wedge \mathrm{Res}(A \wedge F_A) \wedge \mathrm{Res}(B \wedge F_B)$. Thus it suffices to show that when step 5 is reached, the conjunction of the

residues of the three formulas is satisfiable.

If step 5 is reached, F_E, F_A, and F_B all entail the same set of equalities between variables. Let S be this set of equalities, and let T be the set of all other equalities between variables of F. We claim that the interpretation which makes every equality in S true and every equality in T false will satisfy $Res(F_E)$, $Res(A \wedge F_A)$, and $Res(B \wedge F_B)$. For if it does not satisfy, say, $Res(F_E)$, then $Res(F_E)$ would entail the disjunction of all equalities in T. But if this were the case, step 4 would have caused a case split and step 5 would never have been reached.

3. Simplification based on Satisfiability Programs.

In section 3.1 we define what we mean by an *incremental, resettable* satisfiability program. In section 3.2, we describe how such programs can be used to implement a simplification procedure, which is a generalization of a joint satisfiability procedure. In section 3.3 we discuss some aspects of the efficiency of our simplification procedure. In section 3.4 we discuss some of its deficiencies.

3.1 Incremental, Resettable Satisfiability Programs

An *incremental* satisfiability procedure is one which accepts literals one by one and which can determine at any time whether their conjunction is satisfiable. If in addition it can mark its state, accept more literals, and later return to the marked state by "undoing" the literals which were asserted after the mark, it is called *resettable.*

From now on we will assume that all the satisfiability programs we use are incremental and resettable, and that they propagate the equalities and disjunctions of equalities which are entailed by the conjunctions they have received. More precisely, a satisfiability program for a theory $\mathcal{A}$ consists of a global data structure, $CONTEXT_A$, for representing conjunctions of $\mathcal{A}$-literals, together with the following functions for manipulating it.

If Q is the conjunction currently represented by $CONTEXT_A$, then:

$ASSERT_A(P)$ where P is a literal, changes $CONTEXT_A$ to represent $Q \wedge P$. If $Q \wedge P$ is unsatisfiable, $ASSERT_A(P)$ returns **false**. Otherwise, if there are any equalities between variables which are entailed by $Q \wedge P$ but not by Q, then $ASSERT_A(P)$ returns the conjunction of all such equalities. Otherwise, if there are any disjunctions of equalities between variables which are entailed by $Q \wedge P$, then $ASSERT_A(P)$ returns one of these disjunctions. Otherwise, $ASSERT_A(P)$ returns **true**.

$PUSH_A()$ saves the current state of $CONTEXT_A$ without changing the conjunction that it represents.

$POP_A()$ restores $CONTEXT_A$ to the state it was in just before the last call to $PUSH_A()$.

$SIMPATOM_A(F)$, where F is a non-boolean expression or a boolean constant, returns an expression F' equivalent to F in this $CONTEXT_A$. F' is the normal form for F in this context. For example, $SIMPATOM_Z(x + 0)$ returns x and $SIMPATOM_Z(x - y)$ returns 0 if $x = y$ is entailed by Q. ($SIMPATOM_A$ will only be called when Q is consistent).

Any decision algorithm can theoretically be used to implement a satisfiability program with these properties. However, it generally requires considerable effort to construct an efficient satisfiability program from an efficient decision algorithm.

We have implemented satisfiability procedures for $\mathcal{Z}$, $\mathcal{V}$, $\mathcal{L}$, and $\mathcal{E}$. The program for $\mathcal{Z}$ is described in detail in [Nelson 1976]. [Nelson and Oppen 1977] describe the data structure used by all programs except $\mathcal{Z}$, and describes the programs for $\mathcal{E}$ and $\mathcal{L}$. The satisfiability program for $\mathcal{V}$ is trivial: it signals splits whenever it finds a term of the form select(store(...)..); that is, it does the obvious split required by its axiom.

Before giving the simplification algorithm, we define the auxiliary function ASSERT*, which accepts an arbitrary literal, splits it into homogeneous pieces, and calls the appropriate assertion functions. We define it for the case where there are satisfiability programs for two theories $\mathcal{A}$ and $\mathcal{B}$.

In this program, P_E, P_A, P_B, Q_E, Q_A, Q_B are variables containing formulas.

ASSERT*(Q):

1. Divide Q into homogeneous pieces Q_E, Q_A, and Q_B as described in section 2.

2. Set $P_E \leftarrow ASSERT_E(Q_E)$; $P_A \leftarrow ASSERT_A(Q_A)$; $P_B \leftarrow ASSERT_B(Q_B)$

3. If any of P_A, P_B, P_E are **false**, return **false**.

4. If any of P_A, P_B, P_E are disjunctions, return one of these disjunctions.

5. If all of P_A, P_B, P_E are **true**, return **true**.

6. (One or more of the formulas are conjunctions of equalities, and the others are **true**. This step will propagate the equalities.) Set each of the variables Q_E, Q_A, and Q_B to be the formula $P_E \wedge P_A \wedge P_B$, and go to 2.

We define the functions PUSH* and POP*; they call the push and pop functions for each of the satisfiability algorithms. We also assume the existence of a function SIMPATOM* which takes an arbitrary term and simplifies it using the information in $CONTEXT_A$, $CONTEXT_E$, and $CONTEXT_B$. It does this by calling the SIMPATOM functions for these three theories. We omit the details. ASSERT*, PUSH*, POP*, and SIMPATOM* are used by the simplifier as a satisfiability program accepting arbitrary literals of the language.

We have four observations to make about these functions.

First, a term t in an inhomogeneous literal which has been replaced by a new variable v in step 1 of some call to ASSERT* may in a subsequent call be replaced by another new variable w. This is all right, since both t = v and t = w are sent to the relevant satisfiability program.

Second, a record is kept of the individual variables generated as labels for terms in step 1, so that SIMPATOM* can put the literals back together, replacing generated labels by the terms they represent.

Third, it is not necessary to send all the equalities to all the satisfiability programs in step 6. As mentioned in section 2.2, an equality need only be sent to a satisfiability program if both variables in the equality are parameters of the conjunction represented in the program.

Fourth, it is important to note that ASSERT*, PUSH*, POP*, and SIMPATOM* do not form a satisfiability program, since ASSERT* may return a disjunction without doing a case split to determine if any of the disjuncts are satisfiable. It would be possible to change ASSERT* to investigate each branch of the disjunction, but this is not delicate enough to be of use in a simplifier. When a case split occurs for which some, but not all, of the cases are satisfiable, the simplifier needs to know *which* branches of the split are satisfiable. For example, consider the problem of simplifying $x \in \{4, -6\} \wedge x > 0$ to $x = 4$. The simplifier must discover that the satisfiable branch of the split is the one in which $x = 4$. This is why ASSERT* returns the disjunction to the simplifier; the latter does the case split as described in the next section.

3.2 Cond-style simplification

In this section we will use LISP list notation for expressions.

Our simplifier first puts expressions into *cond normal form*. (This is similar, but not identical, to the cond normal form in [McCarthy 1963].) An expression is defined to be in cond normal form if the following holds.

(1) The expression does not contain any boolean connectives other than **cond**. Thus $A \wedge B$ is replaced by the equivalent (**cond** A B **false**), and $\neg A$ by (**cond** A **false true**).

(2) No first argument to a **cond** is a **cond**. Thus (**cond** (**cond** P A B) C D) is replaced by (**cond** P (**cond** A C D) (**cond** B C D)).

(3) No expression of the form (**cond** P A B) is the argument to any function other than **cond**; thus (F (**cond** P A B)) is replaced by (**cond** P (F A) (F B)).

(4) Every boolean subexpression (other than constant subexpressions **true** and **false**) is the first argument to a **cond**. For instance, a single atomic formula P which is not the first argument to a **cond** is replaced by (**cond** P **true false**). F(X=Y) is successively replaced by (F (**cond** (= X Y) **true false**)) and (**cond** (= X Y) (F **true**) (F **false**)).

(In practice, the transformation required by clause 4 is not carried out if the subexpression is a second or third argument to **cond**, since this would waste space. The cond normal form of (**cond** P A B), if A and B are boolean, is (**cond** P (**cond** A **true false**) (**cond** B **true false**)) but we store it as (**cond** P A B). In the discussion below, we assume that the transformation has been made. We will not consider here how to determine whether a subexpression is boolean.)

Cond normal form is not a canonical form, since two syntactically different expressions, each in cond normal form, may be logically equivalent.

An expression in cond normal form corresponds naturally to a binary tree whose nodes are labelled with atomic formulas. We call this tree the *cond tree* for the expression. To the expression (**cond** P A B) corresponds the tree whose root is labelled with P, whose left son is the tree for the expression A, and whose right son is the tree for the expression B. The tree for any non-cond expression E is a node with outdegree zero labelled with E. Thus every node in a cond tree is either an internal node with two sons and a boolean expression as label, or a leaf node whose label is either non-boolean or one of the constants **true** or **false**.

The maximum number of nodes in the cond tree for an expression of length n may be exponential in n. But, by sharing structure, the tree can be represented as a directed graph, and the amount of storage required is linear in n.

Let N be a node of the tree. Then $\langle N_1\ N_2\ \ldots\ N_k \rangle$ is the *branch* to N if N_1 is the root of the tree, $N_k = N$, and, for each $1 \le i < k$, N_{i+1} is a son of N_i. The *context at N* is defined to be the conjunction $L_1 \wedge \ldots \wedge L_{k-1}$, where each L_i is the label of N_i if N_{i+1} is the left son of N_i, and the negation of the label of N_i otherwise.

The context of a node is exactly the condition that must hold for an evaluator to reach the node during evaluation of the expression. That is, if the conditional expression is regarded as a program fragment, the context of a node is the strongest "invariant assertion" on the arc leading to the node. For example, consider the following expression in cond normal form: (**cond** P (**cond** Q A B) (**cond** R C D)). The context of the node for B, that is, $P \wedge \neg Q$, is the condition that B would be evaluated if the whole expression were evaluated.

It follows that the disjunctive normal form of a formula is the disjunction of the contexts of the leaves labelled with **true** in the cond tree for the formula. Cond normal form is much more compact than (traditional) disjunctive normal form because, in the former, disjuncts are represented as branches in a tree (or paths in a directed graph) and thus may share structure.

To simplify an expression, the simplifier traverses its cond tree, maintaining as it does so a representation for the context of the node it is visiting. It ignores nodes whose contexts are inconsistent.

Besides pruning away the branches which are inconsistent, the simplifier collapses together branches to leaves with equivalent labels. If the expression is a valid formula, every leaf which is reached will be labelled **true**; all these branches will be collapsed, and **true** will be returned. Similarly an unsatisfiable formula simplifies to **false**.

The following algorithm simplifies a formula F, which we assume is in cond normal form. NORMALIZE is a function which returns the cond normal form of its argument.

SIMPLIFY(F):

1. If F is not of the form (**cond** P A B), return SIMPATOM(F).

2. F is of the form (**cond** P A B). Call PUSH*(). Set Q ← ASSERT*(P). If Q = **false**, then POP*() and return SIMPLIFY(B). (P cannot be true, so F is equivalent to B in the context in which it appears. Furthermore, the context of B is equivalent to the context of F.)

If Q = **true** then set AA ← SIMPLIFY(A). Otherwise, set AA ← SIMPLIFY(NORMALIZE((**cond** Q A NIL))). (In this case, Q is a disjunction. The third argument to the **cond** is irrelevant, as explained below.) Call POP*(), and go on to step 3.

3. Call PUSH*(). Set Q ← ASSERT*(¬P). If Q = **false**, then POP*() and return AA. (P cannot be false, so F is equivalent to A in the context in which it appears, and the context of A is equivalent to the context of F.)

If Q = **true** then set BB ← SIMPLIFY(B). Otherwise, set BB ← SIMPLIFY(NORMALIZE((**cond** Q B NIL))). Call POP*(), and go on to step 3.

4. If AA = BB, return AA. Otherwise, let P = SIMPATOM(P). If AA = **true** and BB = **false**, return P. Otherwise return the expression (**cond** P AA BB).

Note how propagations are "spliced" into the formula. For instance, suppose that ASSERT*(P) returns a disjunction D ∨ E in step 2. We simplify the normalized form of (**cond** (D ∨ E) A NIL), which is (**cond** D A (**cond** E A NIL)). In simplifying this expression, A is simplified twice, once assuming D and once assuming E. The NIL is never reached, since its context, ¬D ∧ ¬E, is inconsistent with the context of F.

We will now sketch the proof of the completeness of the algorithm. We say that the context of a node is *convex* if it does not entail any disjunction of simple equalities without entailing one of the disjuncts. Whenever the context of its argument is non-convex, SIMPLIFY calls itself recursively on some **cond** expression. Thus whenever its argument is not a **cond** expression, its context is convex. The proof of correctness of the joint satisfiability algorithm shows that if a context is convex, and no satisfiability algorithm has propagated **false**, then it is consistent. Therefore whenever SIMPLIFY returns from step 1, the context is consistent. If F is valid, every leaf of its cond tree with a consistent context is labelled with **true**, so every term returned in step 1 is **true**. It follows by induction that AA and BB are always **true**, and therefore that the algorithm is complete.

3.3 Comparison with DNF-style Theorem Proving

We do not know how to give an adequate analysis of our simplifier; since its behaviour in practice is much better than its worst case behaviour.

Instead, we will compare our approach, using cond normal form, with an obvious alternative approach, using disjunctive normal form, which we call a DNF-style approach. We assume that we are only interested in proving validity of formulas and are not interested in simplifications of arbitrary expressions.

We assume that the formula is represented as a cond tree with n internal nodes.

The most obvious algorithm to prove the formula is to put its negation into disjunctive normal form and to test each disjunct for unsatisfiability. This corresponds to testing that the context of each leaf labelled with **false** is unsatisfiable. The standard DNF-style approach builds up the context for each leaf from scratch, that is, from the root of the cond tree. The number of calls to ASSERT equals the sum, taken over all leaf nodes labelled with **false**, of the length of the branch to the leaf. This sum varies from $O(n)$ to $O(n^2)$, and has an average value of $O(n^{1.5})$, if one considers all binary trees with n internal nodes and all external node labellings with **true** or **false** to be equally likely. There are no calls to PUSH or POP. A non-resettable satisfiability program can be used.

Our algorithm makes n calls to PUSH, n calls to POP, and 2n calls to ASSERT. Therefore, DNF-style algorithms minimize (to zero) the number of calls to PUSH and POP, while our algorithm minimizes the number of calls to ASSERT. To determine which method is better, we would need to know the expected number of calls to ASSERT which each algorithm makes on realistic input distributions and the relative costs of resettable satisfiability algorithms and non-resettable ones.

The formulas which arise in the Stanford Verifier are often implications between conjunctions of literals. (Formulas with this structure arise in program verification whenever the invariant assertion on a simple loop is a conjunction of literals.) If there are n conjuncts in the antecedent of such a formula and m conjuncts in the consequent, then the disjunctive normal form of the negation of the formula has length m(n + 1), while the cond tree has only m + n internal nodes. A DNF-style algorithm can therefore make as many as m(n+1) calls to ASSERT, while our algorithm will make at most m + n calls to ASSERT, PUSH and POP. On this sort of example, our approach seems superior.

3.4 Finding the Simplest Form

In this section, we will note some problems with our present simplification algorithm. These problems do not arise when our simplifier is used as a theorem prover, but only when it is being used to simplify expressions which do not simplify to an atomic symbol such as **true**. These problems arise in the design of any simplification algorithm.

First, a problem common to all normal forms is that they may lose some of the structure of the original expression. It is hard to recover this structure if the expression does not significantly simplify. For instance, using cond normal form, the formula $(A \vee B \vee C) \wedge (D \vee E \vee F)$ is "simplified" to

```
(cond A (cond E true (cond D true F))
    (cond B (cond E true (cond D true F))
        (cond C (cond E true (cond D true F))
            false)))
```

and (**cond** E **true** (**cond** D **true** F)) is duplicated in three places. Our simplifier actually converts this formula back to a formula involving the usual boolean connectives, but the present version of the simplifier does not find the original (and simplest) form of the expression. This has not been a serious problem in our system; it only becomes a problem when the original formula is not simplifiable and is in a form close to conjunctive normal form.

Another problem occurs when simplifying conjunctions like $x \leq y \wedge y \leq x \wedge x = y$. The simplifier discovers that the last equality is redundant and simplifies the conjunction to $x \leq y \wedge y \leq x$ instead of to $x = y$. (Had the equality appeared first, both inequalities would have been removed as redundant.) There does not seem to be any way to handle this problem without extending the set of primitives for manipulating contexts. For example, if a call to ASSERT made earlier conjuncts in the context redundant, this might be detected and exploited. It probably would not be too difficult to modify ASSERT in this manner, but it might create unacceptable complications in the simplification algorithm.

A significant problem concerns implementing the test AA = BB in step [4] of our simplification algorithm. This is intended to collapse branches of the **cond** tree which lead to identical results; for example (**cond** P 1 1) should simplify to 1. If AA or BB are atomic symbols, there is no problem. If they contain **conds**, testing for logical equivalence is possible but probably impractical. If they contain no **conds**, then testing them for equality (using the lisp EQUAL) will usually be sufficient, if SIMPATOM puts expressions into a canonical form. However there is a difficulty: consider (**cond** (= X 1) (F 1) (F X)), which we would like to simplify to (F X). Our SIMPATOM chooses (F 1), not (F X), as the canonical form when X = 1 is known, so in step 4 AA is (F 1) and BB is (F X). A completely adequate test for collapsing the two branches would require testing whether $Q \wedge P$ entailed AA = BB, in which case BB should be returned, otherwise whether $Q \wedge \neg P$ entailed AA = BB, in which case AA should be returned. (Q is the context of F, which is of the form (**cond** P A B).) Again the overhead may be prohibitive. This problem actually arises frequently and is more troublesome in practice than any of the other problems we have mentioned in this section.

4. Notes

The language accepted by the simplifier is richer than that described in section 1. All predicates (including =) and boolean connectives are considered boolean-valued functions (that is, functions which evaluate to the booleans **true** and **false**). Terms are allowed to contain arbitrary boolean-valued expressions. Expressions are allowed as functions. The following simplifications illustrate this generality.

$F(\textbf{true}) \supset F(X \vee \neg X)$;
true;

cond(**true**, F, G)(X)
F(X);

The axioms assumed by our simplifier do not enforce strict typing. For instance, **cons**(X, Y) + **store**(V, I, E) is an acceptable expression (that the simplifier will simplify to itself). We plan to add type predicates (or type constants and a type function) to the next version of our simplifier.

The simplifer does not store conjunctions of atomic formulas as strings or LISP s-expressions, but in a graph with one vertex for each term and subterm in the conjunction. Another data structure is used to represent an equivalence relation on the vertices. Two vertices are equivalent if the terms they represent are known to be equal in this context. To propagate an equality, a satisfiability procedure merges two equivalence classes; this can be done very efficiently. The details of this representation are given in [Nelson and Oppen 1977].

Using this representation, it is not necessary to generate "labels" for terms which appear in inhomogeneous literals.

This representation also allows the efficient implementation of other routines in our simplifier more efficient, such as PUSH and POP. Obviously, one way to implement PUSH would be to have it make a physical copy of the existing context; equally obviously this is not very satisfactory. The approach we take is to keep a *history* of all changes we make to our global data structure; popping then involves undoing these changes until we reach the context of the last call to PUSH.

The simplifier includes a decision procedure for the theory of the rationals, but not for the theory of the integers. In this respect, our simplifier does not differ from most theorem provers. A satisfiability program for the integers would have to be able to determine whether a conjunction of linear inequalities is satisfiable over the integers. This is commonly called the integer programming problem; it is much more difficult in practice than the rational linear programming problem.

Luckily, most formulas that tend to arise in practice (at least in program verification and program manipulation) do not depend on subtle properties of the integers. Further, there are some easily-implemented heuristics (such as converting $x < y$ into $x + 1 \leq y$) which treat integer variables as rationals and work well in practice.

We also wish to handle multiplication in our simplifier (multiplication of two variables: multiplication by a constant is correctly handled). One approach would be to include some heuristics to handle the cases that arise in practice. Another approach, which we prefer, would be to implement a decision procedure for the reals under addition and multiplication.

Our simplifier is not a general purpose theorem prover; it cannot prove quantified theorems of the predicate calculus. However, in the Stanford Verifier, it is used in conjunction with a program called the *rulehandler* which accepts user-supplied lemmas. During a simplification, the rulehandler instantiates the free variables of the lemmas and sends the instantiated lemmas to the simplifier. In our system, the rule handler stands in the same relation to the simplifier as the satisfiability programs. The rule handler can be viewed as a satisfiability program driven by user-supplied axioms.

Acknowledgment

We thank the Stanford Verification group for their patience in waiting two years for this simplifier.

References

[Craig 1957] W. Craig, "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory", Journal of Symbolic Logic, volume 22.

[Downey and Sethi 1976] P. Downey and R. Sethi, "Assignment Commands and Array Structures", manuscript.

[Johnson and Tarjan 1977] D. S. Johnson and R. E. Tarjan, "Finding Equivalent Expressions", manuscript.

[McCarthy 1963] J. McCarthy, "A Basis for a Mathematical Theory of Computation", in Computing Programming and Formal Systems, edited by P. Braffort and D. Hirshberg, North-Holland Amsterdam.

[Nelson 1976] C. G. Nelson, "Documentation for Z", unpublished memorandum.

[Nelson and Oppen 1977] C. G. Nelson and D. C. Oppen, "Fast Decision Algorithms based on Union and Find", Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science, October 1977.

[Oppen 1978] D. C. Oppen, "Reasoning about Recursively Defined Data Structures", Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, January 1978.